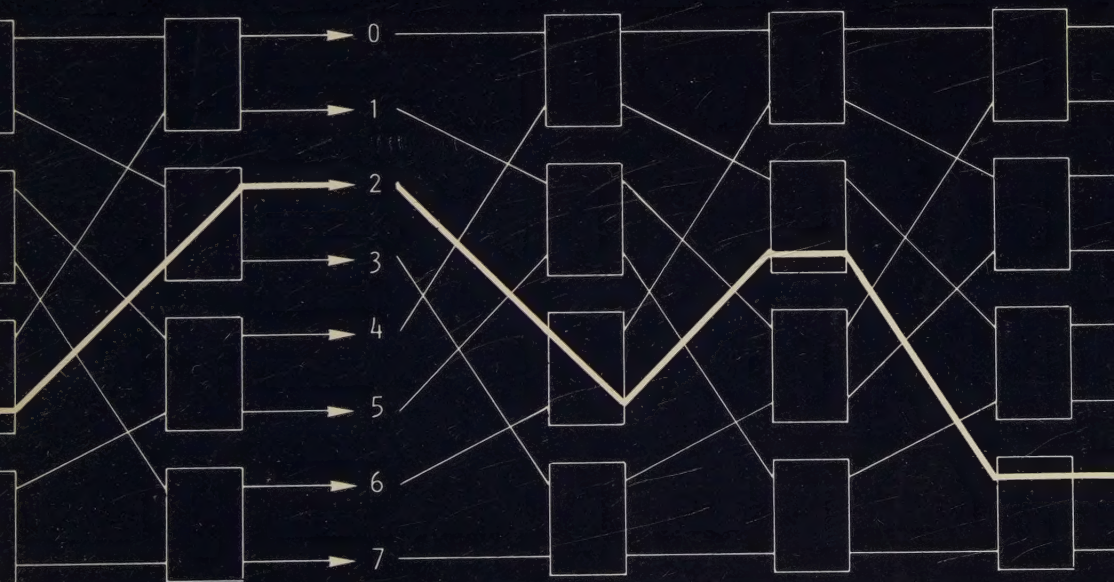


THE LUCAS ASSOCIATIVE ARRAY PROCESSOR AND ITS PROGRAMMING ENVIRONMENT

Christer Fernström



The LUCAS Associative Array Processor
and its Programming Environment

AKADEMISK AVHANDLING

som för avläggande av teknisk doktorsexamen vid tekniska fakulteten vid Universitetet i Lund kommer att offentligen försvaras i hörsal E:B, Lunds Tekniska Högskola, onsdagen den 18 maj 1983 kl 10.15

av

Christer Fernström

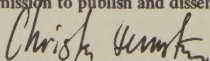
civ.ing.

Organization LUND UNIVERSITY Avd. för Digital- och Datorteknik, LTH Box 725, 220 07 LUND, Tel 046/107515 Dept. of Computer Engineering, Univ. of Lund, PO Box 725, S-220 07 LUND, Sweden		Document name DOCTORAL DISSERTATION	
		Date of issue May 1983	
		CODEN: LUTEDX/(TEDD-1002)/1-285/(1983)	
Author(s) Fernström, Christer		Sponsoring organization	
Title and subtitle The LUCAS Associative Array Processor and its Programming Environment			
Abstract The LUCAS project is an attempt to design and to evaluate a highly parallel system. As part of the project an SIMD organized Associative Array Processor comprising 128 bit-serial processors has been implemented. This thesis reports on the design of LUCAS and of languages suitable for implementation on this and on similar designs. A language for microprogramming has been defined and implemented on LUCAS. The language has a block structure where microprograms are contained in modules together with variables and subroutines they use in common. It features control constructs similar to those of high-level languages (if-then-else, while, repeat). An iterate-statement, which has a very natural application to bit-serial processing has been introduced in the language. The compiler, which produces efficient horizontal microcode is described. For application programming, a high-level language has been defined in terms of extensions to Pascal (Pascal/L). The extensions needed to express parallel processing in an SIMD organized computer include new data types and new constructs in the control structure. The definition of a Pascal/L pseudo-machine suggests how the language could be implemented on LUCAS. The execution of some constructs in Pascal/L on the pseudo-machine are described in detail.			
Key words parallel processing, associative, SIMD, bit-serial processing, microprogramming, compaction, high-level language, p-code.			
Classification system and/or index terms (if any) Computer Science, Electrical Engineering			
Supplementary bibliographical information			Language English
ISSN and key title			ISBN
Recipient's notes		Number of pages 285	Price
		Security classification	

Distribution by (name and address)

I, the undersigned, being the copyright owner of the abstract of the above-mentioned dissertation, hereby grant to all reference sources permission to publish and disseminate the abstract of the above-mentioned dissertation.

Signature



Date

April 19, 1983

Department of Computer Engineering
University of Lund S-220 07 Lund, Sweden

THE LUCAS ASSOCIATIVE ARRAY PROCESSOR AND ITS PROGRAMMING ENVIRONMENT

Christer Fernström



THE LUCAS ASSOCIATIVE ARRAY PROCESSOR AND ITS PROGRAMMING ENVIRONMENT

Christer Fornström



Printed in Sweden
Studentlitteratur
Lund 1983

A Catherine et Nicolas

Contents

PREFACE

PART 1 - INTRODUCTION

<u>Chapter 1. PARALLEL COMPUTERS</u>	15
1.1 INTRODUCTION	15
1.2 CLASSIFICATION OF PARALLEL MACHINES	17
1.3 MORE ABOUT SIMD	19
1.4 PERFORMANCE IN PARALLEL MACHINES	21
1.5 ASSOCIATIVE ARRAY PROCESSORS	24
1.5.1 Associative Memories	25
1.5.2 Bit-serial Working Mode	31
1.5.3 A Bit-serial Associative Processor	33
1.6 INTERCONNECTION NETWORKS IN SIMD SYSTEMS	36
1.6.1 Introduction	36
1.6.2 The Perfect Shuffle	38

PART 2 - LUCAS

<u>Chapter 2. SYSTEM ARCHITECTURE</u>	50
2.1 SYSTEM OVERVIEW	50
2.2 CONTROL UNIT	54
2.2.1 Data Paths	55
2.2.2 Instruction Timing	56
2.2.3 Sequencer	56
2.2.4 Address Processor	59
2.2.5 Common and Mask Registers	63
2.2.6 I/O Buffer Register	64
2.2.7 Status Register	65
2.3 PARALLEL PROCESSING ARRAY	66
2.3.1 Processing Element	66
2.3.2 Communication Between Elements	69
2.3.2.1 I/O Bus	69
2.3.2.2 Interconnection Network	70
2.3.3 Select First Chain	71
2.4 HOST COMMUNICATION	74
2.5 PHYSICAL IMPLEMENTATION	76
 <u>Chapter 3. BASIC SYSTEM SOFTWARE</u>	 78
3.1 INTRODUCTION	78
3.2 SOFTWARE SUPPORT	79
3.2.1 Microprogramming	80
3.2.2 Machine Level Programming	82
3.2.3 High Level Language Programming	84

PART 3 - LUCAS MICROPROGRAMMING LANGUAGE

Chapter 4. LANGUAGE DEFINITION

94

4.1 INTRODUCTION	94
4.2 MICROPROGRAMMER'S VIEW OF LUCAS	98
4.3 INTRODUCTION TO THE LANGUAGE	101
4.3.1 Program Structure	101
4.3.2 Notation	105
4.3.3 Lexical Rules	106
4.4 LANGUAGE ELEMENTS	107
4.4.1 Constants	107
4.4.2 Variables, Assignments	107
4.4.3 Subroutines	110
4.4.4 Microprograms	111
4.4.5 Statements I - Program Flow Control	114
4.4.5.1 General	114
4.4.5.2 Conditions	114
4.4.5.3 If-Then-Else	115
4.4.5.4 While	116
4.4.5.5 Repeat	116
4.4.5.6 Iterate	117
4.4.5.7 Exit	118
4.4.6 Statements II - Array operations	120
4.4.6.1 PE Instructions	120
4.4.6.2 Input and Output	121
4.4.6.3 Common, Mask and I/O Buffer Operations	122
4.5 PROGRAMMING EXAMPLES	124
4.6 CONCLUSIONS	129

Chapter 5. LANGUAGE IMPLEMENTATION

131

5.1 INTRODUCTION	131
5.2 LEXICAL ANALYSIS	133
5.3 PARSER	135
5.3.1 Syntax Analysis	135
5.3.2 Error Recovery	137
5.3.3 Semantic Routines	140
5.3.4 Translation to Intermediate Code	143
5.4 INTERMEDIATE CODE	151
5.5 CODE IMPROVEMENT	153
5.5.1 Packing	154
5.5.2 Compaction	159
5.5.3 Transformation and Pipelining	170
5.5.4 Other Improvements	174
5.6 CONCLUSIONS	179

PART 4 - HIGH LEVEL LANGUAGE PROGRAMMING

Chapter 6. LANGUAGES FOR PARALLEL MACHINES - A SURVEY

6.1 INTRODUCTION	181
6.2 BASIC CHARACTERISTICS	182
6.3 THREE CASE STUDIES	187
6.3.1 Glypnir	187
6.3.2 IVtran	190
6.3.2.1 Language Elements	192
6.3.2.2 The Paralyzer	195
6.3.3 Parallel Pascal	197

Chapter 7. PASCAL/L - A HIGH LEVEL LANGUAGE FOR LUCAS

7.1 INTRODUCTION	201
7.2 LANGUAGE DESCRIPTION	204
7.2.1 Declaration of Data	204
7.2.1.1 Selectors	204
7.2.1.2 Parallel Arrays	206
7.2.2 Indexing of Parallel Variables	208
7.2.3 Expressions and Assignments	209
7.2.4 Control Structure	212
7.2.5 Standard Functions and Procedures	214
7.2.5.1 Data Alignment	214
7.2.5.2 Selector Operations	215
7.2.5.3 Input and Output	216
7.2.6 Microprograms	217
7.3 EXECUTION ON LUCAS	218
7.3.1 Pascal/L Pseudo-machine	219
7.3.2 Parallel Expressions	228
7.3.3 Where Statement	230
7.4 PROGRAMMING EXAMPLES	235
7.5 CONCLUSIONS	241

PART 5 - CONCLUSIONS

<u>Chapter 8. CONCLUSIONS</u>	243
8.1 GENERAL	243
8.2 LUCAS PROCESSOR ARRAY	244
8.2.1 Design	244
8.2.2 The Impact of VLSI Implementation	246
8.3 LANGUAGE ASPECTS	249
8.3.1 Microprogramming	249
8.3.2 Pascal/L	250
APPENDIX 1. MICROPROGRAMMING INSTRUCTION SET	252
APPENDIX 2. SYNTAX OF THE MICROPROGRAMMING LANGUAGE	264
APPENDIX 3. SYNTAX OF THE PASCAL/L EXTENSIONS	272
REFERENCES	275

Preface

We live in a time when computers are becoming an essential part of society and when the number of problem areas where computers can be used is rapidly increasing. This is due to the fast technical development during the last decades, which pushes the state of the art in two directions: computers are becoming more powerful and they are getting smaller. The increase in power comes from both technological progress and from new forms of organization where parallelism between the parts that make up the computer system is a key concept.

The LUCAS project is an attempt to design and to evaluate a system which is highly parallel but still kept within reasonable limits for a university research project. The initial plans, greatly inspired by Caxton Foster [Fos-1], were drawn in 1978 and the project started in the autumn of the same year, involving three graduate students: Ivan Kruzela, Bertil Svensson and myself.

The first part of our research was to study the feasibility of different applications on a loosely defined SIMD-organized associative array computer. The idea was to find one or a few applications to which a more precise design should be directed. This resulted in two applications which seemed suitable for the architecture we had in mind: signal processing in the form of Fast Fourier Transform on real time data and low-level image processing. Algorithms were tested on a computer simulated model of the architecture.

In 1979 a first design of the machine was made and a prototype with eight processing elements, LUCAS I, was completed. Experience gained from this design together with some new ideas, especially concerning the organization of the control unit and of the interaction with the

host computer, resulted in a design of a slightly modified machine, LUCAS (II). The implementation started in 1980 and in the spring of 1982 the machine was fully operable with 128 processing elements.

During the period 1980 to 1982 our work proceeded in different directions. The hardware was debugged and the basic system software was written. At the same time new applications were studied. The use of similar architectures (STARAN) in operations on relational databases had been studied by other people, but their results were not convincing. The discovery that LUCAS could be used efficiently for rather complex operations on relations, as defined by relational algebra, and not only for searching, was an important result.

My interest was directed towards the more general principles of how to program the machine. Given its organization, the question was: what should a programming language look like? Obviously there is no definite answer to this question since it depends on what the user needs. To obtain maximum possible speedup over a sequential computer, it is reasonable that the language is semantically close to the architecture. On the other hand, many users are prepared to trade efficiency against ease of programming. In the thesis some of these aspects will be discussed.

The thesis is divided into four main parts: an introductory part which comprises Chapter 1, the second part which deals with the design of LUCAS and which includes Chapters 2 and 3, the third part which is concerned with the design and the implementation of a microprogramming language for LUCAS and which comprises Chapters 4 and 5, and finally the fourth part, where high-level languages for parallel computers are discussed.

Chapter 1 should be considered as background material and most of the material presented can be found in different text books describing parallel computers. In the text, parallel computers in general and SIMD organization in particular is discussed, the principles of associative memories and associative processors are demonstrated (much of this material is from Foster [Fos-1]). The final section deals with interconnection networks in parallel computers.

Chapter 2 describes the organization of LUCAS. This part of the research was a joint effort between the three persons involved in the project.

In Chapter 3 the programming environment of LUCAS is described. Different levels of programming are discussed: microprogramming of the parallel processors, machine level programming, where the program controls both the host and the parallel processors, and finally the procedure library, which is used to integrate LUCAS operations in Pascal programs, is demonstrated.

In Chapter 4, a medium level language for microprogramming is defined. The language offers a new programmer's view of the machine. It comprises high level like control constructs, parameterized subroutines, local and global variables and a module concept.

Chapter 5 deals with the implementation of the language. The general principles of the compiler are described and the algorithms used for the compaction of the horizontal microcode are presented in some detail. The produced code is approximately of the same quality as code produced by programming in microcode assembly.

Chapter 6 is a survey of programming languages for highly parallel computers. Ideas from several languages are presented and three languages are described in somewhat more detail.

In Chapter 7 a high-level language is defined as an extension to Pascal. The language includes the concept of parallel variables and also extends the control structure of Pascal. The implementation of the language on LUCAS is considered and a parallel pseudo-machine is defined which could execute the pseudo-code produced by a compiler.

In Chapter 8 we summarize our results and give suggestions for further work in the area.

I am indebted to many people for their help during the project and during the preparation of this thesis. I want to express my appreciation to Dr. Rolf Johannesson for his guidance and valuable

support throughout the work. The influences from my two colleagues Ivan Kruzela and Bertil Svensson are present at many pages in the thesis and without the constant exchange of ideas between the three of us, nothing of this would have been possible. I also owe a great deal of gratitude to Lennart Ohlsson for his constructive criticism and suggestions for improving the presentation.

Several people have given valuable comments on the design of LUCAS and of Pascal/L and I want to express my thanks to Hilding Elmqvist, Mats Cedervall, Anthony Reeves, Peter Molin and Anders Ardö. Anders Ardö has also implemented the text formatting system which greatly simplified the work of preparing the manuscript.

Special thanks goes to Lena Månsson for her exceptional skill, care and attention when preparing the drafts. I would also like to thank Andy Ruckemann for his help and encouragement, Dan Leek and Joseph Wajnbloom for their great help with the technical details of the implementation of LUCAS and Björn Breidegard for his never ending encouragement and strong belief in our ideas.

Lastly, I would like to express my deepest appreciation to my wife Catherine Solignac for her continuing support and patience and for her valuable suggestions concerning the organization of the thesis. A final thought goes to our son Nicolas who was born the same day a full scale version of LUCAS with 128 processing elements successfully completed the test programs, thus doubling my pleasure.

Part 1. Introduction

Chapter 1 PARALLEL COMPUTERS

1.1 INTRODUCTION

The rapid development of computers during the last decades has pushed the state of the art in different directions. Due to drastic reductions in size and price of small computer systems, completely new application areas are being exploited. The view of the computer as a system component has been adopted by engineers and designers. At the other end of the spectrum are the so called supercomputers. The supercomputers of today have 100,000 - 1,000,000 times the capacity of the earliest main frames.

Advances in different fields have contributed to the development: the technological progress has influenced speed, cost and size of the components, new algorithms have been developed for the basic operations, such as arithmetic operations, and new forms of organizing the entire systems are used, where parallel operation between the system components is exploited.

All these areas have influenced the development, but the technological aspects have probably been dominating, especially the increase in

speed in gates and flip flops. Unfortunately we are approaching the possible limits and a further push in technology will not result in a large speedup. For instance the CRAY-1, which was introduced in 1976 has a basic clock cycle period of 12.5 ns. When a new version of the CRAY machine was announced in 1982, the speed of the clock had been set to 9.5 ns. This means that the enhancement in circuit technology alone only allows a relatively small gain in speed.

At 12.5 ns the electrical signals propagate 3.75 meters during one clock cycle period. Even if future technology allows an increase of the clock frequency with a factor 100-1000, the communication delays within the system will play a dominating role in the speed of the machine. It can thus be concluded that questions concerning the organization of systems together with the development of new algorithms will play an even more important role in the future.

The most commonly used form of parallelism is instruction pipelining, where the next instruction is fetched during the execution phase of the current instruction. In this way it is possible to make significant increases in speed by allowing simultaneous activities. However, this is not enough to classify a machine as a parallel computer. In a parallel computer the organization allows several data elements to be processed simultaneously. A typical situation is when two multi-element data variables, such as vectors, are combined. For example:

```
for i:=1 to 64 do
    a[i]:=a[i]+b[i];
```

which can be executed in less than 64 instruction cycles on a parallel computer. A parallel computer with 64 processing units can execute this vector addition in one instruction cycle, provided that each processing unit can read its operands and store its result without conflict with the other processing units.

Before concentrating on the kind of parallelism we find in an associative array processor such as LUCAS, we will briefly discuss some different forms of parallel organizations.

1.2 CLASSIFICATION OF PARALLEL MACHINES

In 1966 Flynn presented a classification of computer organization [Fly-1], which is based on the number of data paths that are used to transport data to and from the processor, and the number of instructions which control the processor in each instant. A stream is defined as a sequence of instructions or data items and depending on whether the instruction and the data streams are single or multiple, four categories exist:

SISD - Single Instruction stream, Single Data stream

SIMD - Single Instruction stream, Multiple Data stream

MISD - Multiple Instruction stream, Single Data stream

MIMD - Multiple Instruction stream, Multiple Data stream

Single Instruction/Single Data

A machine in this group is strictly sequential in the sense that only one data element is processed during each time interval. It contains one single processing unit (referred to as a CPU). This organization is found in microcomputers and in most of the earlier minis (the PDP-11 series but not the VAX for example). This is the conventional von Neumann computer.

Single Instruction/Multiple Data

An SIMD computer consists of one control unit together with several processing units. In each basic time interval all the processing units execute the same instruction but on different data. This class of computers is well suited for applications where the same (often rather simple) calculation is performed on a large number of well

structured data elements. Within the SIMD group we find array processors and associative processors.

Multiple Instruction/Single Data

It is doubtful whether it is possible to define an organization where several instructions operate on a data item simultaneously. However, some people put the so called pipelined computers in this category. In the processor of a pipelined computer there is one entry point and one exit point for data. Once a data item has entered the processor, it is moved from one processing unit to the next, each processing unit being controlled by its own instruction stream.

Any operation that can be divided into a sequence of suboperations can use pipelining. Since the data flow through the pipeline is synchronous, it is desirable that the division results in suboperations of approximately the same complexity.

Multiple Instruction/Multiple Data

In this form of organization several processing units execute in parallel, controlled by separate control units, which each executes its own instruction stream in an asynchronous fashion. This class includes all form of multiprocessor configurations.

1.3 MORE ABOUT SIMD

We have described an SIMD machine as a system consisting of a number of processing units, all of them controlled by a single control unit and executing the same instruction on different data in a synchronous way. Although this is a good description of the overall organization of a computer, it is useful to do a further refinement of the concept of SIMD machines. There are two main types of processors in the SIMD category, namely associative processors and array processors. An associative processor is built around an associative memory (or content addressable memory). Certain memory cells can be singled out, or selected, depending on their contents when they are matched against a centrally broadcasted pattern. Reading and writing of data can be done simultaneously in all selected cells. In an array processor, each cell is usually more complex than in an associative processor, and furthermore the cells are interconnected by a more or less elaborate communication network.

Different taxonomies for SIMD computers have been presented, Higbie [Hig-1] suggests the following division:

- * Array processor - a processor that processes data in parallel and addresses the data by address.
- * Associative memory processor - a processor that operates on data in parallel and where data is addressed by tag or value rather than by address.
- * Associative array processor - a processor that is associative and also operates on arrays of data.
- * Orthogonal processor - a processor with two subsystems. An associative array processor and a serial (SISD) processor which share a common memory

Thurber [Thu-1] proposes the following definition of SIMD processors:

- * SIMD processor - a computer architecture characterized by an SIMD orientation of data and procedure streams.
- * Array processor/Parallel processor - an SIMD processor in which the cells usually bear some topological relationship to each other.
- * Ensemble - a topologically unstructured array processor.
- * Associative processor - an SIMD processor in which the prime means of element activation is an associative process. Generally the cells of an associative processor have a loose topological relationship and are functionally very simple. The processor is usually designed around an associative memory system.

We feel that Thurber's taxonomy is the more relevant of the two in that it stresses the use of an interconnection network in an array processor as compared to a processor ensemble. In the following, when we talk about an "associative array processor", we mean an associative processor, as defined by Thurber, where a communication network defines a topological relationship between the processing elements. As proposed by Slotnick [S10-1], we will use the term "processing element", or PE for short, rather than "processing unit", since this suggests a simpler internal structure, as is commonly the case in SIMD systems.

We terminate this section with the observation that the name "array processor" sometimes is used to designate a processor which is "suitable for processing arrays". These "array processors" are usually pipelined back-end computers which are attached resources to minicomputers. In order to make a distinction between the two, terms such as "SIMD array processor" and "parallel array processor" on the one hand and "pipelined array processor" on the other can be used.

1.4 PERFORMANCE IN PARALLEL MACHINES

As we have mentioned earlier, it is the need for larger capacity which is the reason for introducing parallelism in the computer system. Therefore it is important to have accurate methods to decide the influence of different design parameters on the capacity.

Intuitively the term "capacity" means different things to different persons (see for example [Kuc-2]). It could be defined by the response time for a terminal user in a time shared system, it could mean the total number of jobs that the system can handle in one day or it could be related to the total time that elapses from a job is entered in the system until it is finished.

We will limit our discussion to three aspects of capacity which are frequently referenced in the literature, namely the bandwidth, the speedup and the efficiency. By the bandwidth we mean the number of operations that can be performed in the system per time unit. The speedup indicates how much faster a computation is done in the parallel machine than if it was executed on an SISD computer. Efficiency, finally, measures the utilization of the parallelism in the machine for a certain computation.

To obtain a value of the bandwidth, we assume that a computation C consists of n instructions which can be performed simultaneously. We assume further that the instructions are independent and can be executed without any form of interaction. A space-time diagram shows the hardware utilization as a function of time. Figure 1.1 is a space-time diagram for the computation on an array processor with p processing elements, where $p < n$ and t is the time needed to execute one instruction.

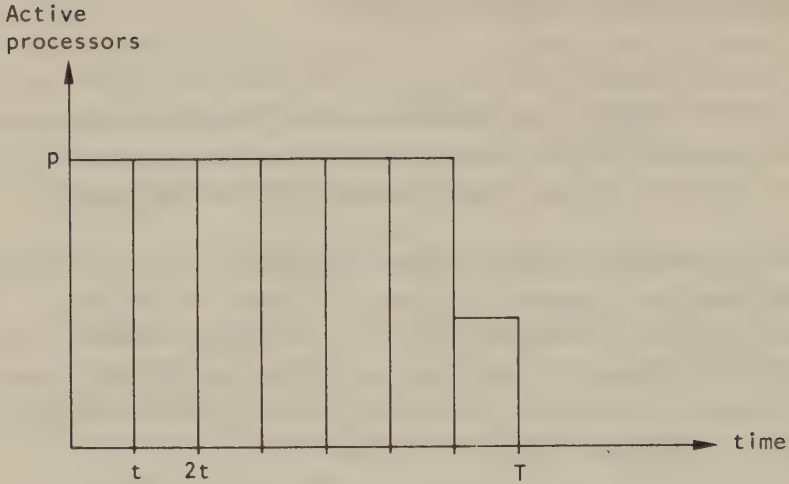


Figure 1.1 Space-time diagram for the computation of C in an array processor with p processors ($p < n$).

If T is the total time to compute C, then we have

$$T = \lceil n/p \rceil * t$$

and the bandwidth, B, which is equal to the number of instructions that are executed per time unit, can be written:

$$B = n / (\lceil n/p \rceil * t)$$

The speedup measures how much faster a computation is performed on a parallel machine than on a sequential machine. First, let the parallelism of a computation be defined as the maximum number of simultaneous subcomputations into which the computation can be divided. Assume that the computation C is partitioned into n steps. In step i the parallelism is $w(i)$ and the time to complete this step on an array processor with p elements, where $p > w(i)$, is $t(i)$. Let $T(i, p)$ denote the time for step i on an array processor with p processing

elements, then we have:

$$\begin{aligned} T(i,p) &= t(i) && \text{when } p \geq w(i) \\ T(i,p) &= \lceil w(i)/p \rceil * t(i) && \text{when } p < w(i) \end{aligned}$$

for the SISD machine we have:

$$T(i,1) = w(i) * t(i)$$

We obtain the speedup by dividing the total time needed on an SISD machine with the total time needed on the array processor and get:

$$s = w(i) * i(i) / T(i,p)$$

We see that the maximum speedup is p and that this is obtained when $w(i)$ is a multiple of p for all i .

The efficiency is simply given by the actual speedup divided by the maximum possible speedup:

$$e = s/p$$

An important question when using a parallel computer is how the set up time influences the total speedup of a computation. By set up time, we understand the time needed to initialize the parallel processor plus the time needed to perform all the non-parallel operations. We split the computation into two parts, one sequential part and one parallel part with the parallelism w . The total computation time is T and the time of the sequential part is t . Then we get:

$$s = \frac{t + w * (T - t)}{t + \lceil w/p \rceil * (T - t)} = \frac{t/T + w * (1 - t/T)}{t/T + \lceil w/p \rceil * (1 - t/T)}$$

Figure 1.2 shows the total speedup as a function of the ratio t/T (the percentage of the total time that is used for set up) for array processors with different number of processing elements. It is assumed that the parallelism of the parallel part of the computation = 128. Figure 1.2 illustrates the great impact any sequential part of an

algorithm has on the speedup of the computation.

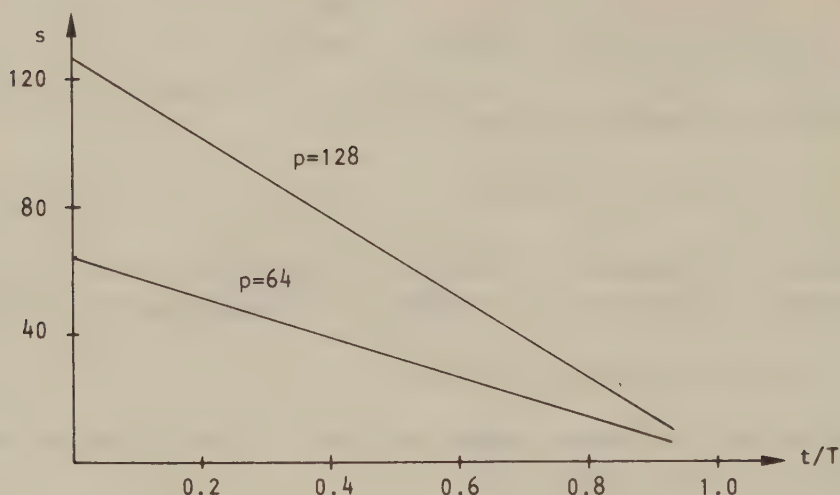


Figure 1.2 The speedup as a function of the t/T ratio for different sizes of array processors.

1.5 ASSOCIATIVE ARRAY PROCESSORS

In Section 1.3 we defined an associative array processor (AAP) as a machine with the following properties:

- * SIMD processor, i. e. it consists of many processing elements (PEs) and one common control unit.
- * Activation of the PEs is an "associative process". This means that it is not the location or address of a PE that decides if it will execute the instructions from the control unit, but some property of the contents in the memory of the PE.

- * A communication network defines a topological relationship between the PEs. They can for example form a linear array or a two-dimensional grid.

In the following we will sometimes use the term associative memory word as a synonym for processing element. We will start our description of AAPs by describing an associative memory, then we will form an associative processor by adding computing capabilities to each word of the associative memory and finally we will present different networks for data communication between the words.

1.5.1 Associative Memories

An associative memory, or content addressable memory, is a memory where each bit cell contains the necessary logic to compare the stored information bit with a search argument. Associated with each word is a one bit result register, called the Tag. Outside the memory are three registers of the same size as the width of the memory words: the Common (or Comparand), the Mask and the Data registers (see Figure 1.3).

To perform a parallel search, the search argument is loaded into the Common Register and a bit map indicating the bits which will be included in the search is put in the Mask. Before the search operation is initiated, all Tag registers in the words that are to be searched are set to ONE. Now a search is performed by generating the two signals "Match 0" and "Match 1" (see Figure 1.4) in the bit positions where the Mask contains a ONE. A mismatch in one bit position generates a mismatch signal that propagates through the OR-gates and finally causes the corresponding Tag to be set to ZERO (see Figure 1.3).

After a search has been executed it is possible to write data to or read data from the selected words - that is the words where the Tag contains a ONE. When writing into the memory, the contents in the

Data register is copied to all selected words, whereas a read yields the logical OR in each bit position of the selected words.

Most likely a unique selection is wanted. This is why a so called multiple match resolver is needed. In its simplest form this is a select first chain, which - when activated - deselects all words except the first one where the Tag contains a ONE (see Figure 1.5). The output from the select first chain can be used as a Some/None indicator (even when the select first function is not activated) and shows if a search was successful or not. This implementation of a multiple match resolver suffers from being slow with one gate delay per word. In the description of LUCAS we will see how the select first chain can be speeded up with the use of look-ahead. A general scheme for high speed multiple match resolvers is found in [And-1].

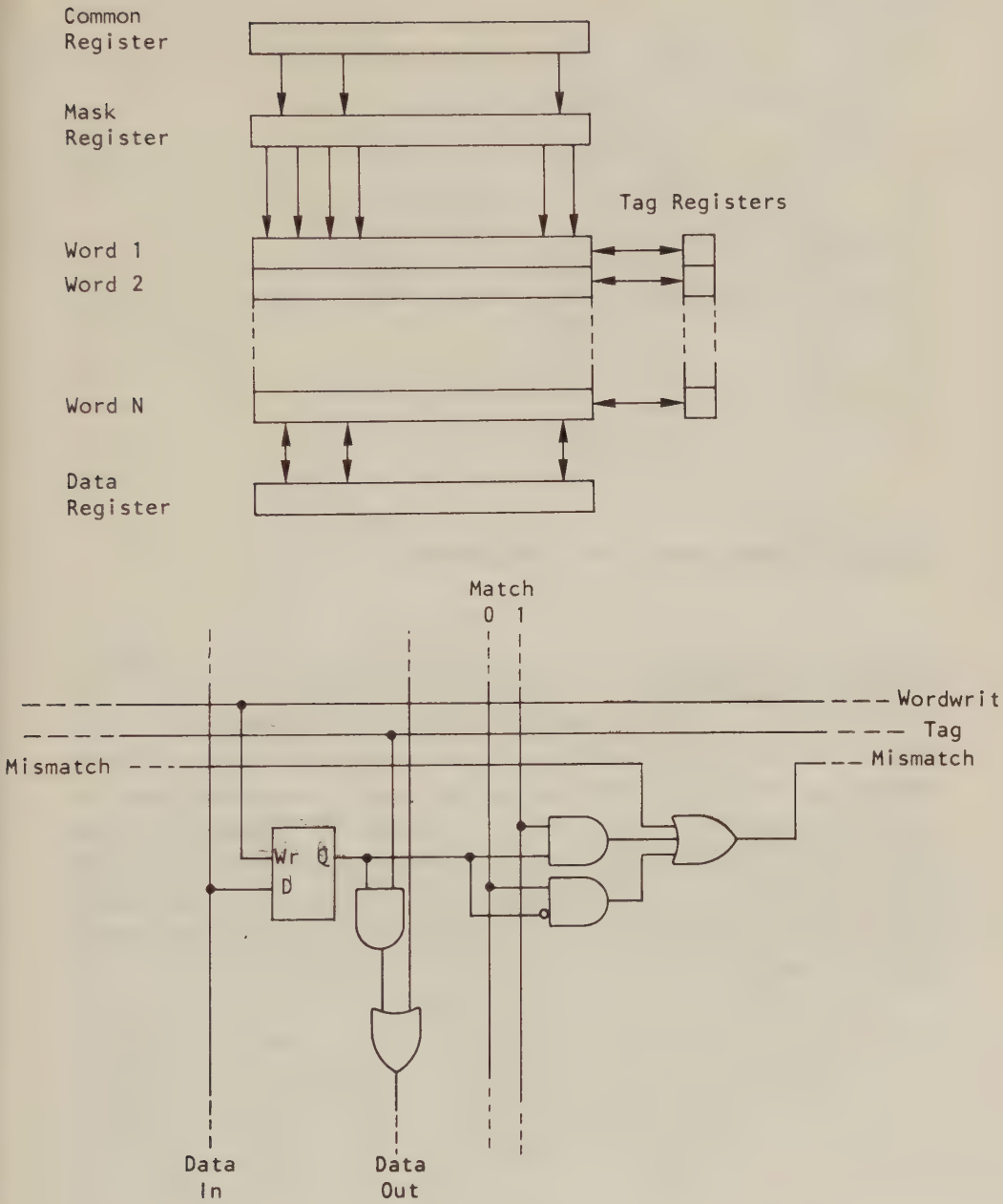


Figure 1.3 Associative memory with detail showing one memory cell.

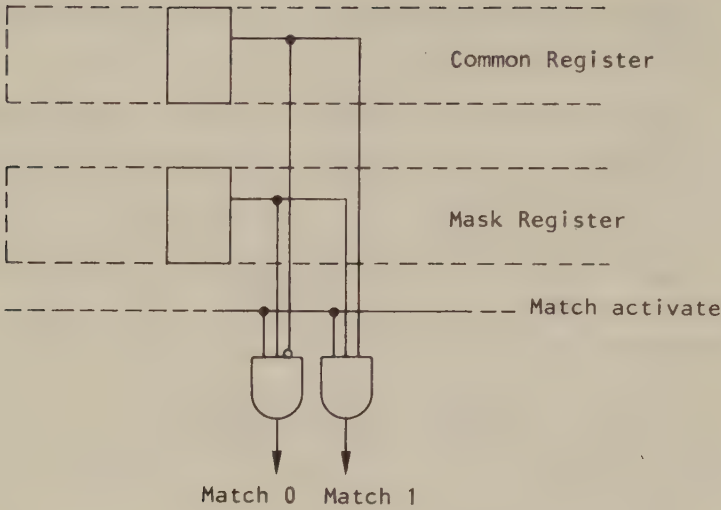


Figure 1.4 Generation of the match signals

In some applications it is desirable to count the numbers of responders to a search operation. Depending on the size of the memory, the speed requirements and the accuracy wanted, different methods may be used. If the size of the memory is moderate and the speed requirements are low or if it is possible to perform the count simultaneously with other activities, one way would be to connect the Tag registers in the form of a shift register, and to shift the Tags n steps, where n is the number of words in the memory, counting the number of ONES.

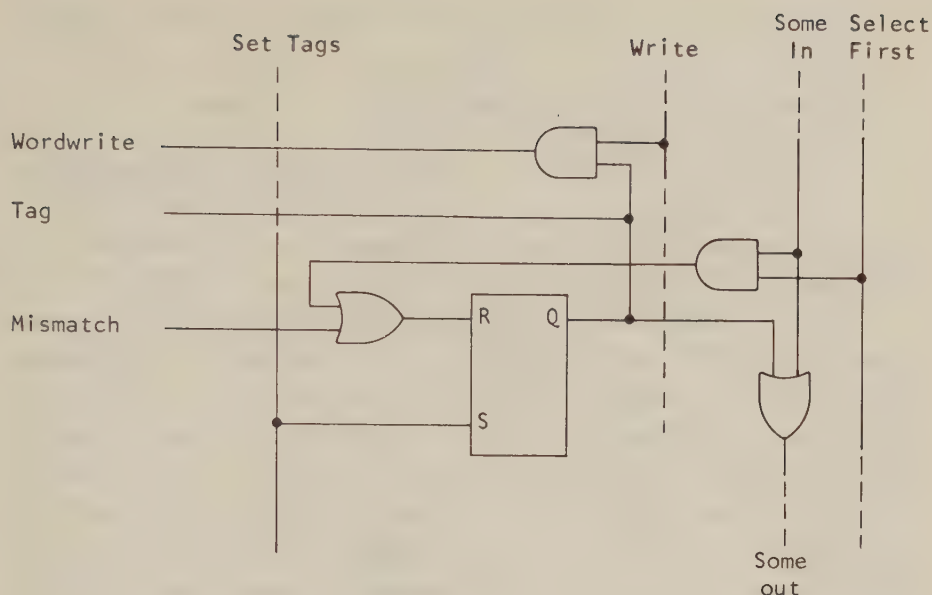


Figure 1.5 The Tag register with a multiple match resolver in the form of a select first chain, which also generates a Some/None signal from the last word.

Another possible way to calculate the number of activated words is with the use of an adder tree (see Figure 1.6). In the first level, groups of tags are processed in combinatorial counters which produce the number of active tags in each group. Assume that g groups are formed, each comprising t tag signals, where g is a power of two. An adder tree of $\log(g)$ levels is then used to produce the total sum. For example, in an associative memory with 4096 words, 16 tags are grouped together, producing 2^8 4-bit numbers after the first level. To produce the total sum, an 8 level adder tree is used.

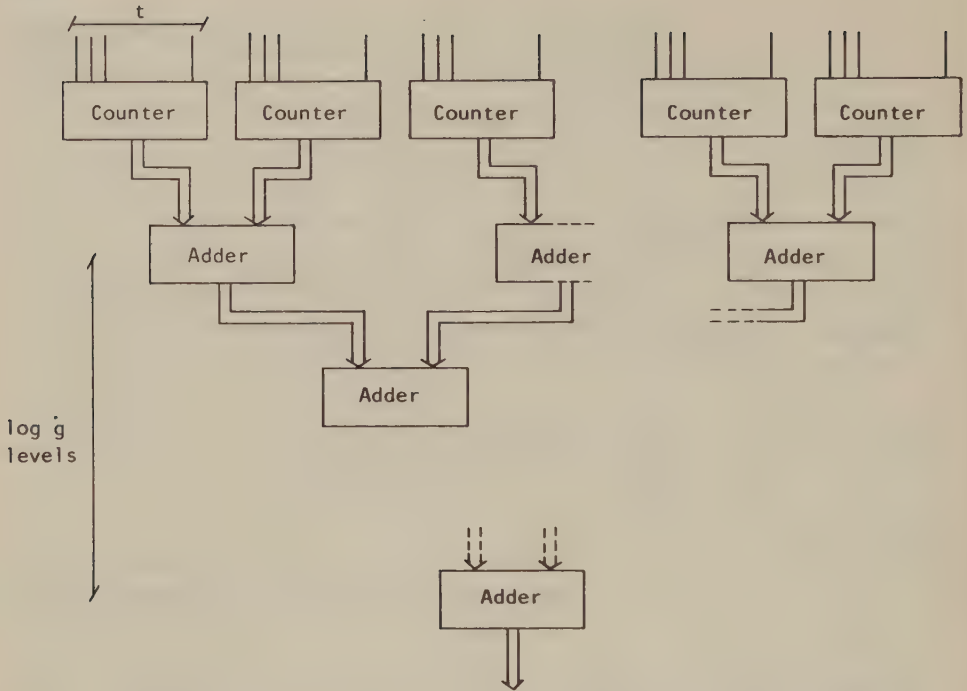


Figure 1.6 Count responders network in the form of an adder tree.

A presumably much faster and much cheaper, although less accurate, method is to use an analog summation circuit to add the current from the active tags. Several other methods are described in [Fos-1].

If the words in the associative memory also can be accessed by address it might be desirable to calculate the address of a responding word. Anderson [And-1] proposes a tree-like structure with $\log(n)/2$ levels of so called p-generators to produce the result. An elegant method, however, would be to reserve $\log(n)$ bits in each word, where the word's address is stored. After a search is performed, followed by a multiple match resolution, a readout gives

the desired address in the reserved bit positions.

1.5.2 Bit-serial working mode

Recalling Figure 1.3 we see that the complexity of the memory cell in an associative memory is only slightly higher than that of a cell in a memory of random access type. What prevents an efficient implementation is the number of connections to the cells. This makes it possible to integrate only a very limited number of cells per IC package. A way to overcome this problem is to reduce the parallelism in the memory. What we have described can be referred to as a "word-parallel-bit-parallel" memory. If we change the design to a "word-parallel-bit-serial" memory, we lose some speed, but on the other hand this allows integration of the memory in a more efficient manner. As compared to an ordinary random access memory (RAM), we will still have a great advantage, since the number of words that are processed in each computation step largely exceeds the number of bits per word, in applications where this kind of organization is used.

Now the memory consists either of shift registers or of RAMs that are accessed "perpendicular" to their normal usage, i. e. an n word by b bit RAM is used as b words by n bits in the associative memory (see Figure 1.7). In the bit-serial working mode, the Mask can be used to decide which bit positions, or bit slices, that are to be processed. Bit positions where the Mask contains a ZERO do not have to be processed at all; which means that the speed of the bit-serial associative memory is not reduced by a factor b as compared to the bit-parallel implementation, but by a factor according to the length of the search argument. A discussion of bit-serial working mode is found in [Bat-4].

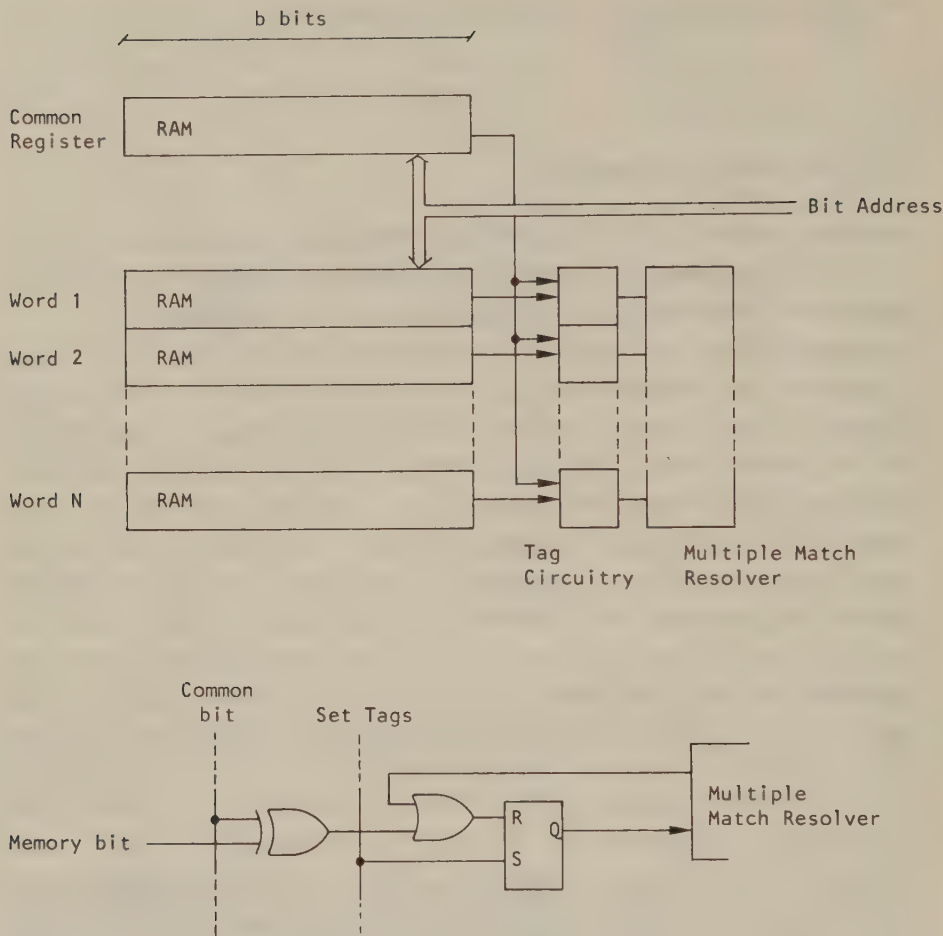


Figure 1.7 Bit-serial associative memory. Each RAM is a b words by one bit memory.

1.5.3 A Bit-serial Associative Processor

It is easy to see how the bit-serial associative memory can be transformed into an associative processor by adding bit processing capabilities to the Tag/mismatch circuitry. This can take the form of a few additional registers and a Boolean function generator (or ALU) which reads the contents of the registers, the PE-memory bit and a bit from the Common register and produces the new register values.

In addition to the Tag register, a one-bit result register (R-register) and a carry register (C-register) is a minimum for implementing bit-serial arithmetic operations with reasonable efficiency. Figure 1.8 shows a possible implementation of a simple PE. It comprises a RAM, an ALU and three one-bit registers: T, R and C. Input of data to the PE memory is handled by the Common register. The data passes from the common input of the ALU and is loaded into the R register, from where it may later be directly written into the memory. Output of data is as follows: the memory output is copied into the R register via the ALU. When the READ signal is activated, the data output shows the value of the logical OR function between the R registers of all selected PEs.

A multiple match resolver is implemented in the form of a Some-None/Select-First chain. A central control unit sends the RAM address, the function code to the ALU and controls the Read, the Write and the Select First functions.

Examples of the ALU functions are:

Set	Set all Tags
Mismatch	Reset Tag if mismatch between the Common and the PE memory bits
Add	Add PE memory bit to R and C. Put result in R (result bit) and C (generated carry)

Load R

Load the R register from the PE memory

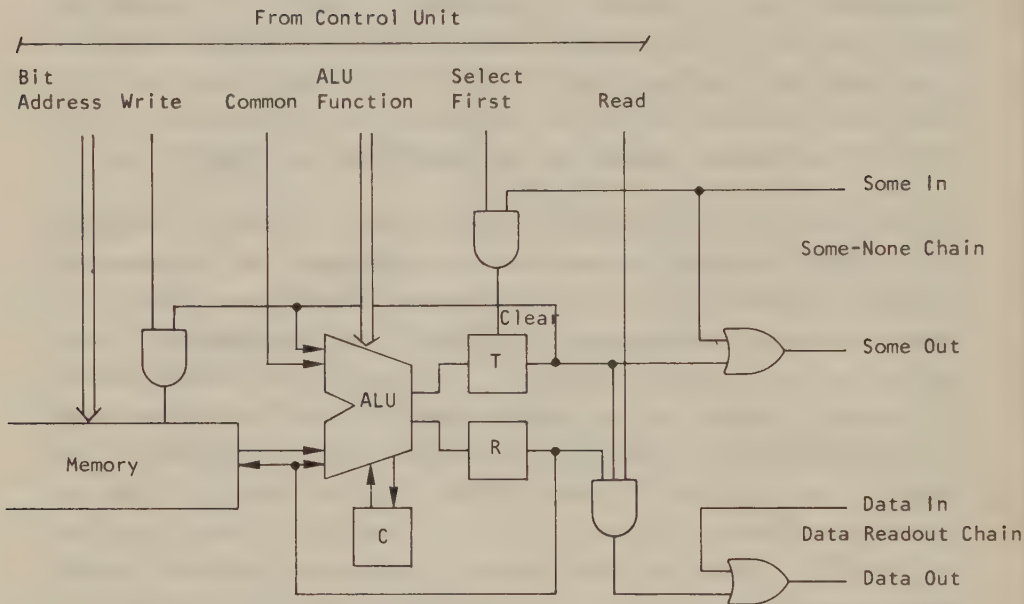


Figure 1.8 Processing element in a bit-serial associative processor

The control unit plays an important role in a bit-serial implementation of an associative processor. It should be able to accept instructions that operate on data items of various sizes, such as 16-bit integers, bytes or ASCII characters and during the execution it should generate the necessary sequence of control signals to the PEs. We call the data items of various formats fields, each field occupying one or several bit slices in the associative memory (see Figure 1.9). So the task the control unit performs is basically a transformation of operations on fields to operations on

bit slices.

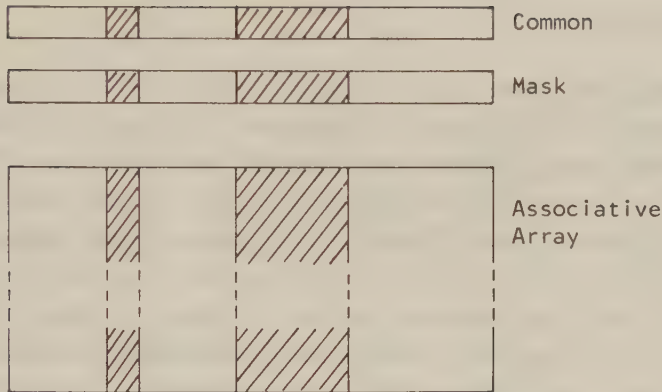


Figure 1.9 Bit slice and Field in the associative memory

The control unit, which may be microprogrammed, contains a Common register, a Mask register, two or three loop counters, a number of registers to hold the addresses of different bit slices in the associative memory, and logic to load and increment these registers.

An instruction to the control unit includes specification of the location of the operand fields (their bit slice address) and also the length of the operands. Examples of such instructions are:

ADDCONST	Add the contents of the Common register to a field
ADDFIELDS	Add the contents of two fields
SEARCH	Compare the contents of the Common register with a field
MAX	Select the word which has the maximum value in the specified field

1.6 INTERCONNECTION NETWORKS IN SIMD SYSTEMS

1.6.1 Introduction

In many algorithms there is a need to combine array elements with different indices. In the case of sequential operation this causes no problem. Since the operands are accessed one after the other, the same bus can be used to transport data, even if the operands are located in different memory modules. The problem which arises in an SIMD machine is that in order to keep as many processing elements busy as possible, the transportation of data must be done in parallel so that all the processing elements have simultaneous access to the data they need. For this purpose, an interconnection network between the processing elements is defined. Once the network has been decided, the topology of the processing array has also been settled.

Since only a small subset of all possible interconnections can be included, the choice of these must be guided by the intended applications. In low level image processing, for example, the most frequently used operations combine the value of a pixel (picture element) with the values of its nearest neighbours to form the new value. In operations on relations in a relational database some algorithm may search two relations with different search keys and then combine the tuples where the search result was affirmative. In signal processing applications, where operations such as the Fast Fourier Transform are common, other ways to combine the operands are needed.

The most simple form of interconnection is when communication is limited to the nearest neighbours. Figure 1.10 shows this form of interconnection in both a one-dimensional and a two-dimensional

array. When communication between the elements is needed, the control unit issues an instruction such as READ ABOVE. Now, each active processing element reads the data coming from the neighbour which is situated immediately above in the processing array.

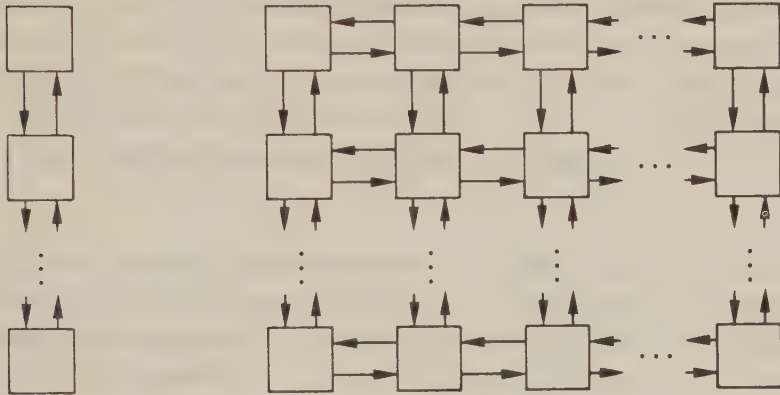


Figure 1.10 One- and two-dimensional processing arrays

Several machines using the one-dimensional interconnection scheme have been designed:

- * RAP, the Relational Array Processor [Ozk-1], which is used for relational database management.
- * PEAC, a Peripheral Array Computer [Sch-1]. Proposed applications are computer assisted tomography and high speed solution of nonlinear partial differential equations.
- * PROPAL 2 [Cim-1], which is used in image processing and general purpose array computing.

This is also one of the interconnection schemes which are used in LUCAS.

Most of the array processors that have actually been built use an interconnection network where neighbours communicate in a two-dimensional array. ILLIAC IV [Bar-1] is configured as an 8 by 8 processing array, the ICL DAP [Red-1] as a 64 by 64 array and the MPP [Bat-5] as a 128 by 128 array. In the case of ILLIAC IV, it is true that the topology defined by the network is a quadrant of 8 by 8 processing elements. However, experience have shown that the size of this processing area is too small for most of the problems and that it is more often used as a one-dimensional array with 64 processing elements where the i :th PE communicates with the elements $i+1$, $i-1$, $i+8$ and $i-8$. The latter two are used to speed up long "shifts" of data.

The advantage of using an interconnection scheme where only neighbours can communicate is of course that it is fairly easy to implement. Physically, all connections can be made short and there is no need for a large bus since all the data movements are local. On the other hand, for many applications the communication is not efficient. It has been shown that the time required to perform an arbitrary permutation in the two-dimensional array is of the order $O(\sqrt{N} \log N)$ [Orc-1].

1.6.2 The Perfect Shuffle

Most of the more elaborate schemes for data communication within an SIMD computer are based on the perfect shuffle. The usefulness of the perfect shuffle was demonstrated by Stone in a "classical" paper from 1971 [Sto-1]. By then, the perfect shuffle had been studied by Golomb [Gol-1] and Pease had shown how it could be used in an efficient calculation of the Fast Fourier Transform on a parallel computer [Pea-1]. Batcher later showed how the Flip Network in STARAN [Bat-2, Bat-3] could be redrawn in the form of a perfect shuffle network [Bat-1].

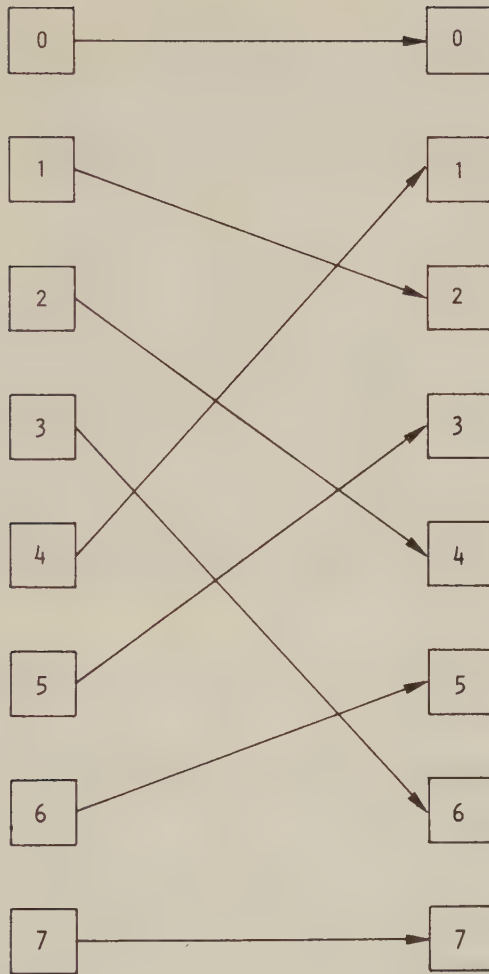


Figure 1.11 The Perfect Shuffle of an 8 element input vector

Figure 1.11 shows the perfect shuffle connection between 8 input lines and 8 output lines. We see that the shuffle, S , permutes the input with index i ($i = i_2 \cdot 2^2 + i_1 \cdot 2 + i_0$) according to:

$$\begin{array}{ll}
 S(i) = 2*i & 0 \leq i \leq 3 \\
 & 2*i-15 & 4 \leq i \leq 8
 \end{array} \quad (1)$$

or, in the general case where we have N inputs and N outputs and $N=2^m$ for some integer m ($i=i_{m-1}*2^{m-1} + i_{m-2}*2^{m-2} + \dots + i_0$):

$$\begin{array}{ll}
 S(i) = 2*i & 0 \leq i \leq N/2 - 1 \\
 & 2*i+1-N & N/2 \leq i \leq N-1
 \end{array} \quad (2)$$

We can see from (2) that the index of an output element, o , can be obtained by rotating the index of the input element, i , one step to the left. This is true since in the case where $0 \leq i \leq (N/2)-1$, $o=2*i$ which is equal to a one-step shift of i and since the most significant bit of i is ZERO, it is also equal to a rotation of i . In the second case, where $N/2 \leq i \leq N-1$, the most significant bit of i is ONE. Thus $2*i-N$ is equal to a left shift of i where only m ($m=\log N$) bits are kept. If we rotate i one step, we get $2*i-N+1$.

It is now clear that after m shuffles, all the elements will return to their initial positions.

One important property of the perfect shuffle is its ability of pairing different operands. Assume we use an array processor with a pairwise interconnection between the processing elements, as in Figure 1.12. Then it is possible to perform parallel operations on pairs of elements whose indices differ in bit 0 only. After one shuffle of the elements, pairs of elements whose indices differ in bit $m-1$ have come in the right position for parallel operation. After $m-1$ shuffles, all pairs of elements whose indices differ in their binary representation in one bit, have been in position for parallel operation in the array processor. This scheme is important for the implementation of the Fast Fourier Transform on LUCAS [Fer-2, Sve-3].

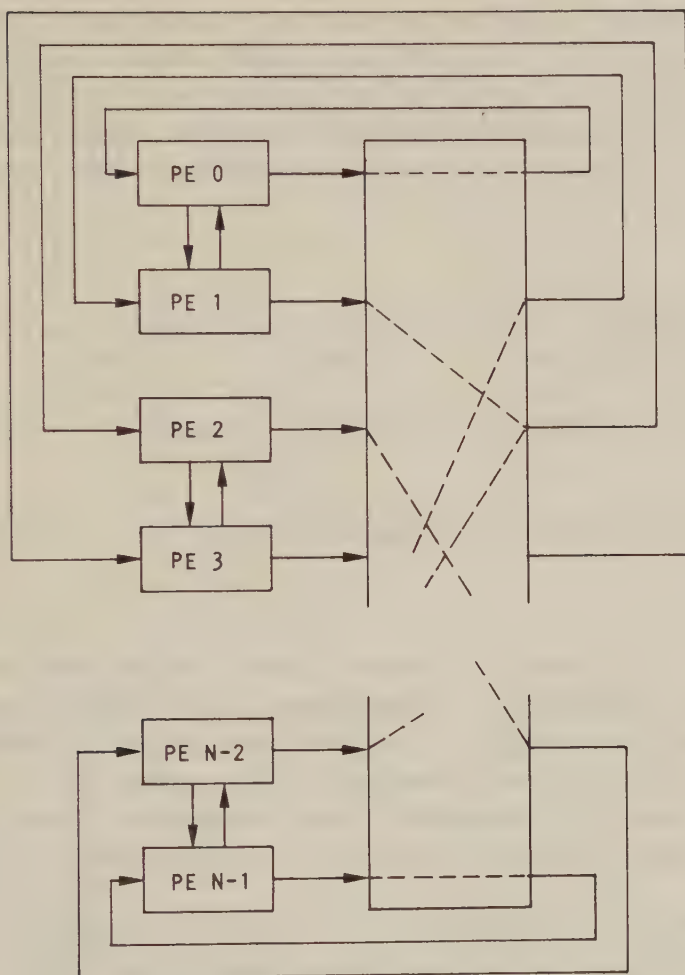


Figure 1.12 N processing elements with a pairwise and a perfect shuffle interconnection

A drastic improvement of the capabilities of the perfect shuffle interconnection is the inclusion of exchange elements (see Figure

1.13) which allows a pairwise interchange of the outputs from the shuffle network. As we have seen, only $\log(N)$ different permutations of the original input vector can be obtained with a pure shuffle connection. (Each shuffling of the data rotates the indices one step so that the original order is restored after $\log(N)$ shuffles.)



Figure 1.13 Permutations in an exchange element

Depending on the value of a control signal, the exchange element either gives a straight or an interchanged data connection between its two inputs and outputs. If we allow each of the $N/2$ exchange elements to be controlled independently of the others, it is in fact possible to obtain any permutation of the N element input vector after a finite number of passes through this shuffle/exchange network.

Shuffle/exchange type networks of this kind can be implemented in the form of a single stage interconnection network (see Figure 1.14), where the input vector is loaded into a register and recirculated through the shuffle/exchange interconnections with different settings of the exchange control signals, until at last the desired permutation is obtained in the register. A multi-stage network consists of several cascaded single-stage networks with or without registers between the stages. If the network includes registers between each stage, data can be pipelined through the network which means that the asymptotic behaviour is like one pass through a single-stage network.

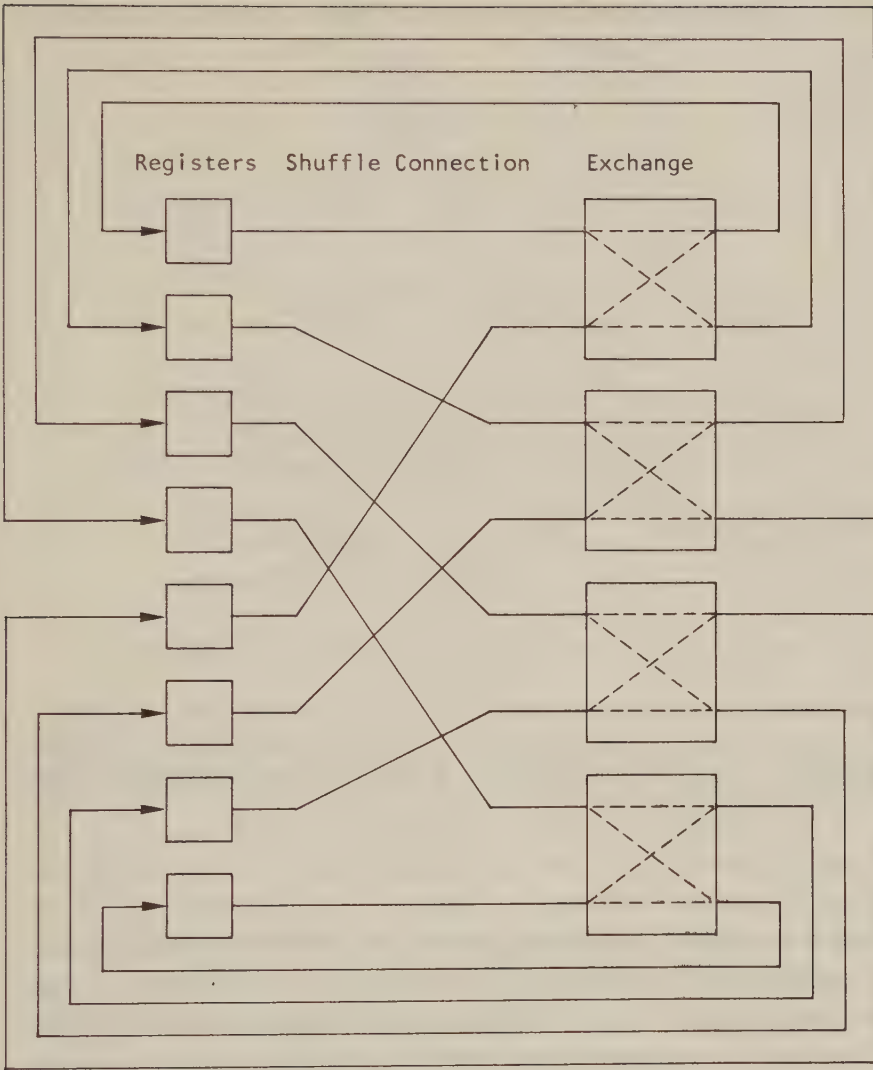


Figure 1.14 One-stage shuffle/exchange network. $N=8$

In [Law-1] Lawrie analyses the performance of the Omega network,

which is a shuffle/exchange network with $\log(N)$ stages (see Figure 1.15).

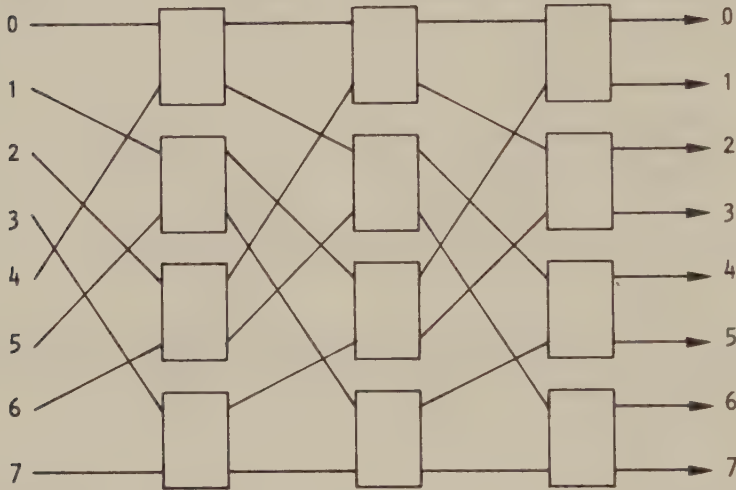


Figure 1.15 The Omega network - a $\log(N)$ -stage shuffle/exchange network. $N=8$

He finds that although the Omega network is not capable of performing all possible $N!$ permutations of an N element input vector in one single pass, it is a very powerful interconnection scheme in that many important permutations can be accomplished such as uniform circular shifts (rotations) and flip-permutations, where one or several bits of the indices are inverted (see Figure 1.16).

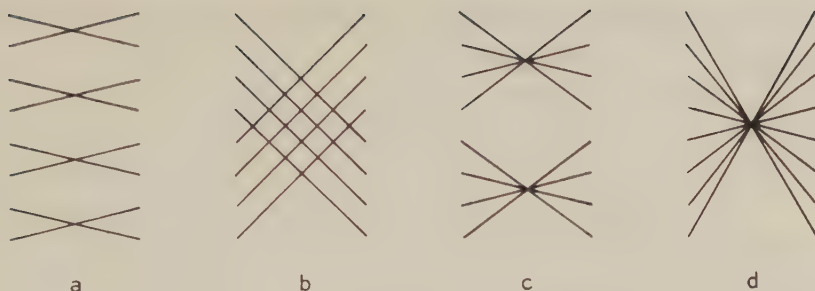


Figure 1.16 Examples of flip-permutations. One or several bits in the binary representation of the indices of the input vector are inverted ($i = i_2 \cdot 2^2 + i_1 \cdot 2 + i_0$). a) i_0 inverted, b) i_2 inverted, c) i_1 and i_0 inverted, d) i_2 , i_1 and i_0 inverted

A simple algorithm is used to calculate the settings of the exchange elements in the Omega network. Assume that the element S of the input vector should end up in position D of the output vector and let $S = s_{m-1}s_{m-2} \dots s_0$ and $D = d_{m-1}d_{m-2} \dots d_0$ be the binary representations of their indices. In the first shuffle stage S is shifted to position $s_{m-2}s_{m-3} \dots s_0s_{m-1}$. Now the exchange element is used to place S in the position $s_{m-2}s_{m-3} \dots s_0d_{m-1}$ by using the bit d_{m-1} of the destination address to control the exchange element: if $d_{m-1}=0$ then S should be connected to the upper output of the exchange element and if $d_{m-1}=1$ to the lower output.

At the inputs of stage i in the network, S has been switched to position $s_{m-i}s_{m-i-1} \dots d_{m-1}d_{m-2} \dots d_{m-i+1}$. It is shuffled to position $s_{m-i-1}s_{m-i-2} \dots d_{m-1}d_{m-2} \dots d_{m-i+1}s_{m-i}$ and exchanged to position $s_{m-i-1}s_{m-i-2} \dots d_{m-1}d_{m-2} \dots d_{m-i+1}d_{m-i}$.

Figure 1.17 shows the settings of the exchange elements for $S=010$ and $D=110$. At the first and second stages the lower outputs are

selected ($D=110$) and at the third stage the upper output is selected ($D=110$).

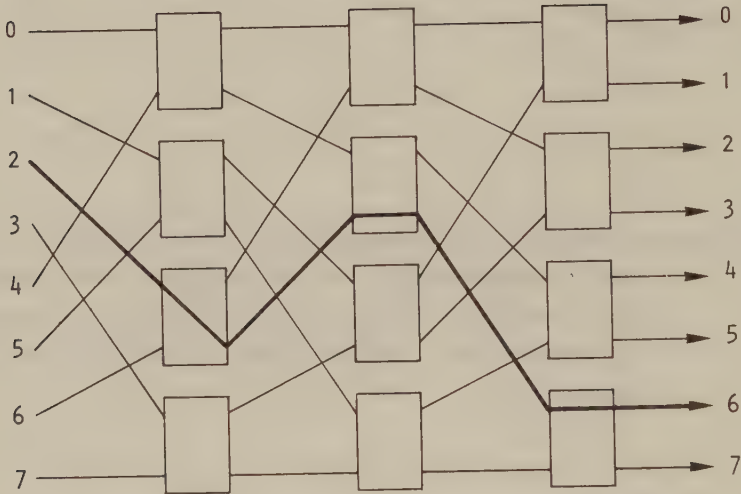


Figure 1.17 The element in position 2 (010) is switched to position 6 (110)

When the Omega network is used to permute all the N elements of the input vector simultaneously, conflicts occur when the two data elements arriving at some exchange element requires connection to the same output. Lawrie showed the properties of a permutation which are necessary for a conflict free solution and proved that many useful permutations have these properties.

Several people have come up with schemes for enhancing the performance of the Omega network.

Lang [Lan-1] starts from a one-stage shuffle/exchange network (as in Figure 1.14). Lawrie's algorithm for the setting of the exchange elements which was presented for a $\log(N)$ -stage network, can of

course also be used in a one-stage network since this can be seen as a time multiplexed variant of the Omega network. In the simple shuffle/exchange network, Lang replaces the exchange elements and the registers by queues, or FIFO registers (see Figure 1.18). Instead of exchanging data elements that appear on the outputs of the shuffle connection, they are placed in the corresponding queue. The difference, as compared to the Omega network, is that both inputs to the exchange stage can be connected to the same output, or rather, more than one data element can be placed in the same queue during a basic permutation step. This corresponds directly to the conflict situation which limits the possibilities of the Omega network.

Each basic permutation step, which is equivalent to one stage in the Omega network, is performed during one or several time steps. In each time step elements are taken from the head of the queues, shuffled and appended to the queues. Associated with each place in a queue is a one-bit flag whose value indicates if the corresponding data element should be shuffled during the current permutation step. A permutation step starts by setting all the flags TRUE and ends when no queue has a TRUE flag in its head element.

Lang shows that the maximum length of the queues is $O(\sqrt{N})$ and that the number of time steps needed to perform an arbitrary permutation also is $O(\sqrt{N})$.

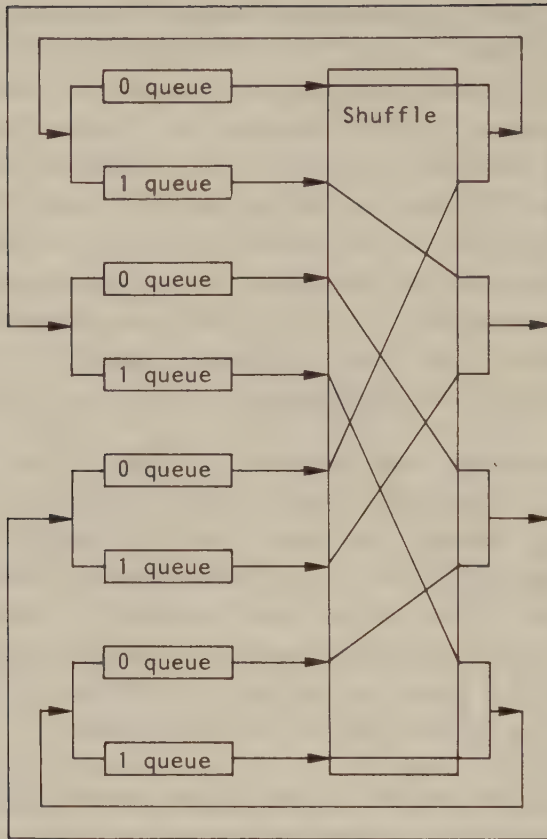


Figure 1.18 The modified one-stage shuffle/exchange network

In [Par-1] Parker shows that three passes through an Omega network is enough to perform any permutation of the input vector and that six passes is enough for any connection (elements of the input vector could be broadcasted to several elements in the output vector). He also claims that two passes might be enough for permutations – it has been shown for $N=4$ and a computer simulation for $N=8$ gives the same

result.

Unfortunately no simple algorithm for the settings of the exchange elements has been found to perform permutations in multiple passes through the Omega network.

Part 2. LUCAS

Chapter 2 SYSTEM ARCHITECTURE

2.1 SYSTEM OVERVIEW

LUCAS consists of three parts: the Control Unit, the Processing Array and the Host. The Host, which is interchangeable, is an ordinary mini- or microcomputer. It takes care of user interaction, filehandling and input/output. Tasks such as program editing, compiling, assembling, loading and printing can also be handled by the Host. Presently a Prolog Z80 microcomputer system is used which includes 64 kbyte of primary memory, 2.5 Mbyte of secondary memory (floppy disks), a graphical display processor, a console terminal and a printer with graphic capabilities. Via a 9600 baud serial link it is connected to a VAX 11/780 and to the LUNET local terminal switching network at the University of Lund.

LUCAS falls into the category of bit-serial-word-parallel machines. During one clock cycle each PE reads or writes one bit of data to the memory. A PE (see Figure 2.1) is built around a local bus through which the different parts of the PE communicate: an I/O register for input and output of data, a 4096 bit memory, a one-bit arithmetic logic unit with four internal registers, and the interconnection logic which gates data from an interconnection network onto the bus.

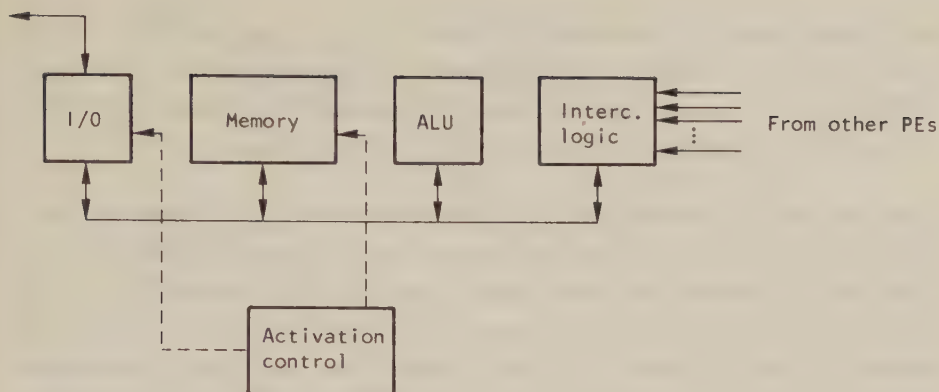


Figure 2.1 Schematic view of a processing element in LUCAS

The activation control consists of a Tag register, which among other things controls the memory so that when the PE receives a WRITE instruction, this has effect only if the Tag has the value ONE.

The interconnection network defines the topology of the Processing Array. On LUCAS, the interconnections are reconfigurable and can be changed by rewiring a strapping area on the circuit boards. Several alternative connections can be defined since the interconnection logic has 7 inputs. Presently the processing elements form a linear array where neighbours can communicate (Figure 1.10) and where data can be moved according to a perfect shuffle/exchange scheme (see Section 1.6.2).

In LUCAS, the Processing Array also forms an associative memory, with each PE being one word in the memory and the Tag holding the result of a parallel search. To allow efficient use of LUCAS as an associative memory, a multiple match resolver, in the form of a select-first chain, is included. It is possible to read out data from

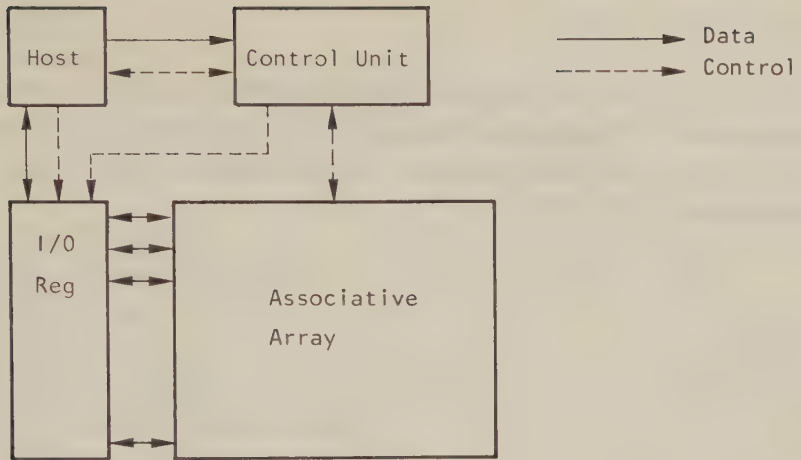
and write data to Tag-selected words. In some applications, e. g. when LUCAS is used as a workspace for operations on relational databases, it is convenient to have this conceptual view of LUCAS as an associative memory with distributed computing capacity.

The role of the Control Unit is to receive instructions from the Host and to execute these on the Associative Array. Since the PEs work in a bit-serial fashion, the most obvious task for the Control Unit is to translate operations on data items (e. g. 8,16 or 32 bit words) to sequences of bit operations. The Control Unit is microprogrammable with a microprogram memory of 4 k words. A word in the microprogram memory is 80 bits wide and microprogramming is horizontal, in the sense that each bit or group of bits in the control word always controls the same part of the hardware. This organization allows parallelism to a great extent between micro operations on different parts of LUCAS. Operations in the Control Unit, such as loop control, can be performed simultaneously with operations in the PEs.

The instruction that the Control Unit receives from the Host consists of an instruction code (8 bits) and a maximum of four parameters (12 bits). The parameters specify the location and the length of the operands within the PE memory. For example:

```
INTADDT  SOURCE1,SOURCE2,DEST,LENGTH
```

The operation INTADDT stands for integer addition in the PEs where the Itag is TRUE. SOURCE1 and SOURCE2 are the parameters that give the address to the least significant bit of the two source operands, DEST indicates the destination address and LENGHT is the length of the operands, e. g. 16,32 or 64 bits or maybe 14, 17 or 27 bits - one advantage with bit-serial computation is the ability to use unorthodox data formats, where the precision is decided by the application and not by the computational hardware at hand.



Figur 2.2 The LUCAS system: Host, Control Unit and Associative Array

Figure 2.2 shows how the three parts of LUCAS are interconnected. Access to the I/O registers is shared between the Host and the Processing Elements. Data to be loaded into the Associative Memory is moved to the I/O registers using a DMA transfer in the Host. Because the I/O register area is mapped into the primary memory space of the Host, it may serve as buffer space for the disk so that data can be transferred directly without the need for intermediate buffering in the Host memory. When loading of the I/O registers is completed, the Host issues an INPUT FROM I/O REGISTERS instruction to the Control Unit.

2.2 CONTROL UNIT

Figure 2.3 gives an overview of the LUCAS Control Unit. Basically it consists of two parts: an interface to the Host and a microprogrammable execution unit which commands the bit-serial processing in the PEs.

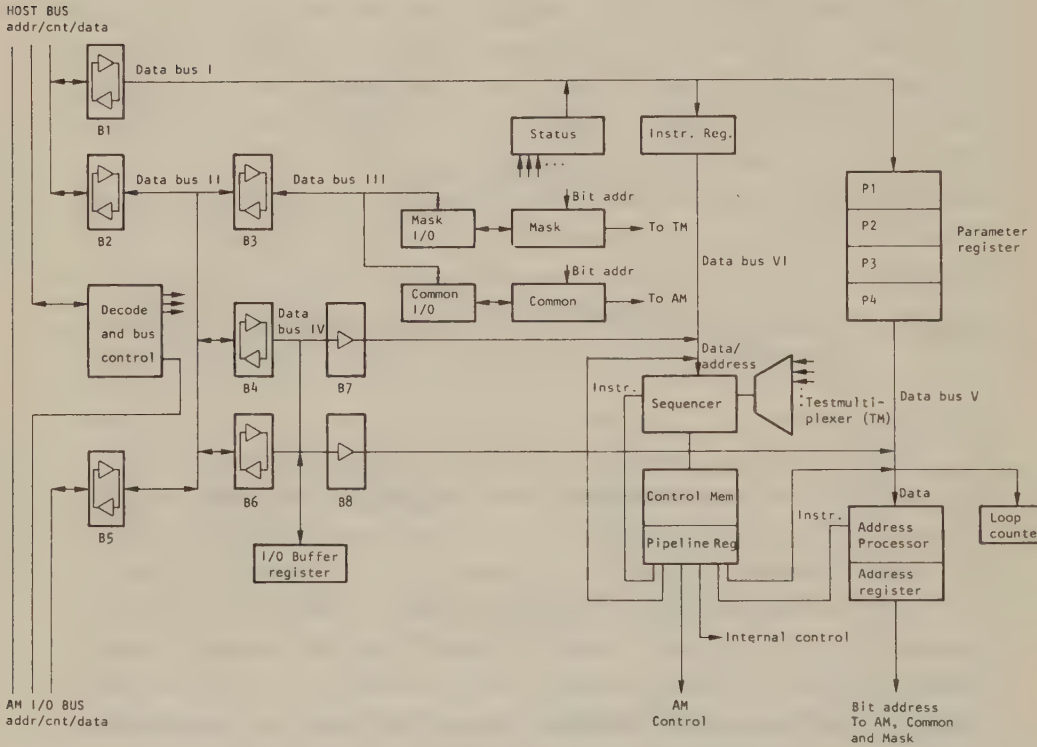


Figure 2.3 The Control Unit. Overview

2.2.1 Data Paths

Communication between LUCAS and the Host is done via registers which are mapped into the memory space of the Host. The registers are:

- Instruction Register
- Parameter Registers (four)
- Status Register
- I/O Buffer Register
- Common I/O Register
- Mask I/O Register
- PE I/O Registers (one in each PE)

To facilitate a change of host computer, one single interface is used which is part of the Control Unit - data to be loaded to or retrieved from the PE I/O Registers passes through the Control Unit on Data Bus II.

In order to support simultaneous activities, several buses are used for communication within the Control Unit, between the Host and the Control Unit and between the Control Unit and the I/O Register area in the Associative Array. The buses are connected to each other through data buffers B1-B8, which are controlled either by the Host (buffers B1, B2, B3 and B4) or by the Control Unit (buffers B6, B7 and B8) or by both (buffer B5).

As seen by the Host, LUCAS occupies a memory area of 512 bytes. The Decode and Bus Control logic senses the status of the Host address bus and when the Host issues a memory request to this area, an acknowledge signal is sent back to the Host in order to disable any ordinary memory that occupies the same address space, and in the Control Unit a communication path is set up through the data

buffers.

2.2.2 Instruction Timing

Prior to the execution of a microprogram, an instruction from the Host is stored in the Instruction Register with parameters in the Parameter Registers. When the Host tries to write data into any of these registers, the Bus Control logic senses if the Instruction Register is empty, in which case data is loaded through Data Bus I and the Host gets an acknowledge signal. If the Instruction Register already contains an instruction whose execution has not yet started, the Bus Control logic sends a non-acknowledge signal which results in the Host entering a wait state. As soon as the Instruction Register becomes free, the new instruction is loaded and the non-acknowledge signal is replaced by a memory write acknowledge. The Instruction Register together with the Parameter Registers are referred to as the instruction pipeline.

To protect the contents of the Parameter Registers from being overwritten, the Instruction and Parameter Register area is not released automatically when the execution of a new microprogram starts, but is controlled via the microprogram - as soon as the parameters have been taken care of.

2.2.3 Sequencer

The role of the Sequencer is to generate the addresses to the Control Memory. At the beginning of a cycle, the current microinstruction (control word) is loaded into the Pipeline Register. The microinstruction contains an instruction to the Sequencer, which, responding to this instruction, generates the address of the next microinstruction in the Control Memory.

The control word is divided into several fields, each controlling one part of the hardware. In this way, the Sequencer itself is controlled

by three fields: one field defines the instruction to the Sequencer, one field specifies the address in case the instruction is a branch and one field selects a test condition to be used for conditional branching. The Testmultiplexer gates different signals to the test input of the Sequencer:

- * The current value of the Mask Register output (see Section 2.2.5)
- * The state of the Busy flag, which indicates if the Instruction Register has been loaded (see Section 2.4)
- * Zero Address Processor status (see Section 2.2.4)
- * Zero Loopcounter status (see Section 2.2.4)
- * Some/None status from the PEs (see Section 2.3.3)

Figure 2.4 shows the Sequencer, which is an Am2910. (For details of this bipolar LSI circuit from Advanced Micro Devices see [Mic-1].) The complete instruction set of the Sequencer can be found in Appendix 1.

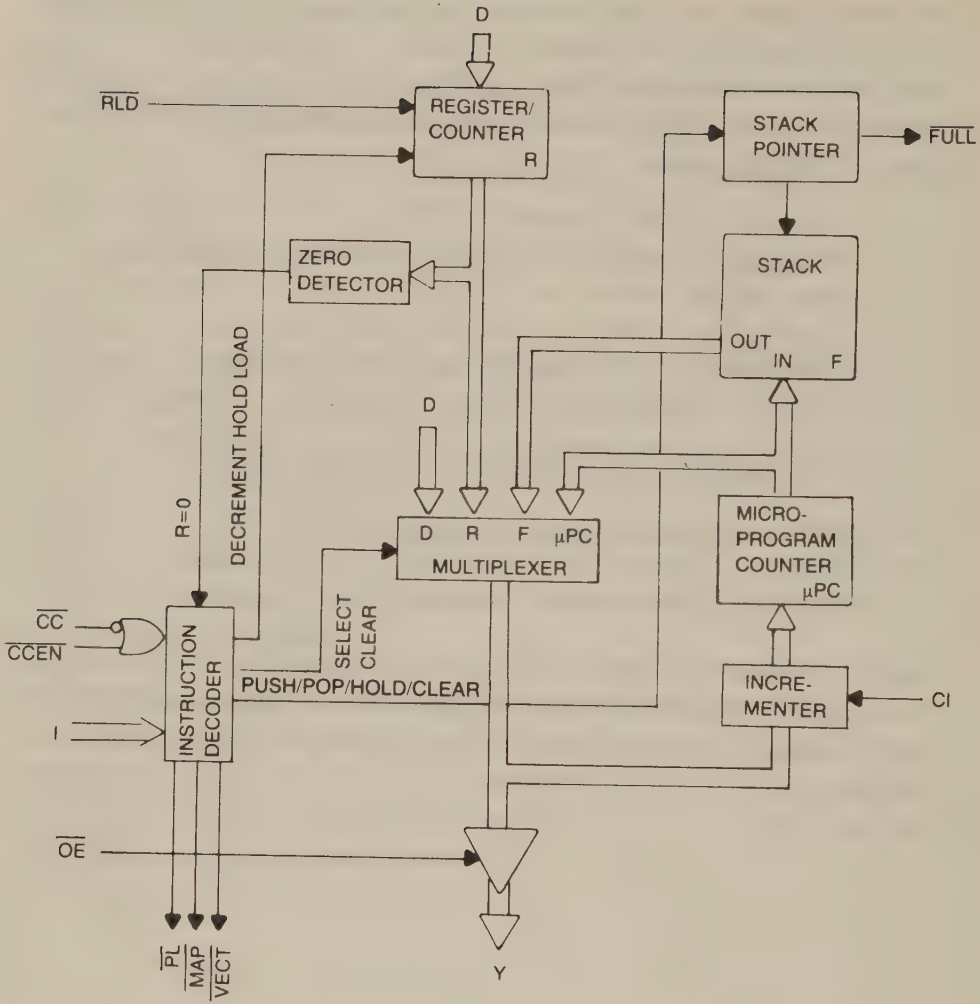


Figure 2.4 Am2910 Block Diagram

2.2.4 Address Processor

A 16 bit field of the microinstruction is used to control the activity in the Associative Array: operations performed in the ALU:s, the set-up of the interconnection network, input/output etc. Since all computations are bit-serial, it is important that the Control Unit can generate the bit addresses needed at a high rate. In the case of the INTADDT (integer addition) operation, the Control Unit needs to generate the following sequence of bit addresses:

```

source1 - bit 0      / used to fetch bit 0 of source 1
source2 - bit 0      / used to fetch bit 0 of source 2
destination - bit 0  / used to store the result bit 0
source1 - bit 1
source2 - bit 1
destination - bit 1
.
.
.
```

The part of the Control Unit which handles this is the Address Processor. The Address Processor is a 12-bit processor capable of doing the kind of integer arithmetic needed for address computations, and to generate a new address each clock cycle. The output from the Address Processor is stored in the Address Register which is used to access the Associative Memory while the next address is being calculated. In this way the Address Processor introduces no delay in the execution of microprograms.

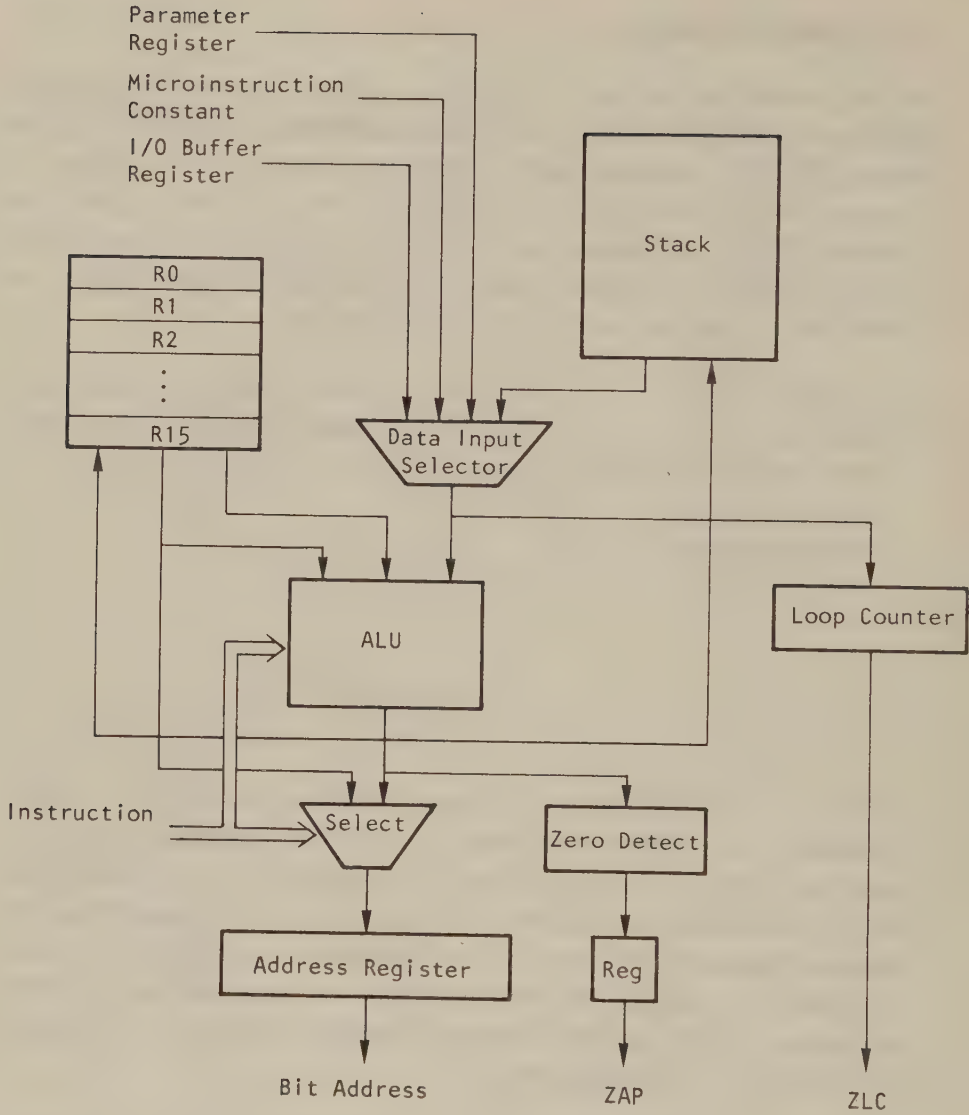


Figure 2.5 The Address Processor

The Address Processor is implemented in the form of three 4-bit

bit-slice microprocessors (Am2901A) and an external data memory organized as a LIFO stack. It has 16 internal registers and an ALU. The stack holds 32 12-bit values. Operands to the ALU are the internal registers, the LIFO stack and data presented on the data input (Data Bus V, see Figure 2.3).

Operations can be performed on:

- any single operand
- any pair of internal registers
- any register and the top element of the stack
- any register and the data presented on the data input

The result of an operation is always stored in the Address Register. It may also be written back into one of the internal registers and it can be pushed on the stack.

In order to reduce the number of bits needed to control the Address Processor, a subset of sixteen instructions have been chosen and a PROM is used to code a 4-bit field of the microinstruction into the 10 bits needed to control the Am2901A. In addition to the instruction to the Address Processor, the microinstruction contains two 4-bit fields which independently selects two of the internal registers for operands (REGA and REGB).

Figure 2.6 summarizes the instruction set of the Address Processor. In the figure, "REGB" stands for the internal register addressed by the REGB field of the microinstruction and "(REGB)" for the contents of this register. "Data" stands for the value of the data inputs.

Mnemonic	Internal Operation	Output
TBZ	None	(REGB)
LDBD	Data $\rightarrow$ REGB	Data
LDBA	(REGA) $\rightarrow$ REGB	(REGA)
INCB	(REGB)+1 $\rightarrow$ REGB	(REGB)+1
DECB	(REGB)-1 $\rightarrow$ REGB	(REGB)-1
ADAD	(REGA)+Data $\rightarrow$ REGB	(REGA)+Data
TAEQB	None	(REGB)-(REGA)
ADBA	(REGB)+(REGA) $\rightarrow$ REGB	(REGB)+(REGA)
SUBBA	(REGB)-(REGA) $\rightarrow$ REGB	(REGB)-(REGA)
AINCB	(REGB)+1 $\rightarrow$ REGB	(REGA)
ADECB	(REGB)-1 $\rightarrow$ REGB	(REGA)
AADAD	(REGA)+Data $\rightarrow$ REGB	(REGA)
AADBA	(REGB)+(REGA) $\rightarrow$ REGB	(REGA)
ASUBBA	(REGB)-(REGA) $\rightarrow$ REGB	(REGA)
ALDBD	Data $\rightarrow$ REGB	(REGA)
PLADR	None	Data

Figure 2.6 Address Processor Instructions

An instruction from the Host to the Control Unit often includes parameters which indicate the location of data within the PEs. These parameters are stored by the Host in the Parameter Registers and once the execution of the instruction has started, the contents of the Parameter Registers can be loaded in to the internal registers of the Address Processor through the data input. In addition to the Parameter Registers, data to the data input can come from several sources: the LIFO stack, a reserved field of the microinstruction or from the I/O Buffer Register (see Section 2.2.6).

A zero-indicator senses an all-zero output from the Address Processor and generates the ZAP (Zero Address Processor) signal, which is fed into the Testmultiplexer. This allows the use of the internal registers as loop counters for microprogram looping.

Since the inner loops of bit-serial arithmetic operations tend to keep the Address Processor busy in each clock cycle (recall the example of the INTADDT instruction), an extra loopcounter has been incorporated in the Control Unit. This 12-bit counter is loaded from the same sources as the data input to the Address Processor. In this way it is possible to load it directly from a Parameter Register to be used in unnested loops. In nested loops the initial loopcounter value is held in one of the internal registers of the Address Processor. To set up the innermost loop, this value is pushed on the stack and then popped back into the loopcounter. Loopcounters for the outer loops are handled internally in the Address Processor.

2.2.5 Common and Mask Registers

The Control Unit also includes a Common and a Mask Register. They are of the same size as the PE memories, 4096 bits, and are addressed by the contents of the Address Register. Input and output of data is handled through their corresponding I/O Registers which are 8 bit shift registers, also accessible from the Host. When an I/O Register has been loaded from the Host, the data is moved into the Common or Mask Register under microprogram control. The output from the Common Register is sent to the PEs on the COMMON line. The Mask output is connected to the Testmultiplexer.

In operations where one of the operands is a scalar, for example in parallel search operations or when adding a constant to a field in the Associative Array, the Common Register is used to hold this operand. The role of the Mask Register is to mask out certain bits when the operations are performed. In bit-serial operations, the Sequencer can use the mask value to conditionally skip certain addresses by "short-circuiting" the loops.

The need for a Mask Register in the Control Unit is not evident because its task can apparently be handled by the Address Processor. However, the fact that data from the Associative Array can be loaded

into the Mask Register during the execution of a microprogram makes it useful in several algorithms, for example:

Multiplication. If we want to multiply all the elements of a field with the same value, the value is moved to the Mask. Multiplication can now be performed using an algorithm which skips over strings of 0's and 1's, resulting in a significant speed-up over the standard algorithm (adding and shifting in each step), which is normally used on LUCAS. (Note that this is particularly important on a bit-serial computer, since addition to the partial product takes much longer time than simply shifting it).

2.2.6 I/O Buffer Register

An important communication link in the Control Unit is the I/O Buffer Register. It is used to move data from a selected PE to the Control Unit. First one PE should be selected. This is accomplished by either a complete associative search or by loading the Tag Registers with a bit slice from the Associative Memory, where the result of a previous search may be stored. To complete the operation, the Control Unit activates the Select First chain, in order to make the selection unique, and sends an IORS (I/O Read Selected) signal to the PEs. The selected PE puts the 8 bit contents of its I/O Register on the I/O Data Bus. The I/O Buffer Register will be loaded through the buffers B5 and B6 (see Figure 2.3)

Once the value has been loaded into the I/O Buffer Register it can be used for several purposes:

- * The contents of the I/O Buffer Register may be written into the I/O Registers in all the PEs. This possibility permits a very flexible communication pattern between the PEs: one associative search operation selects a data source, a second search selects one or several destinations.

- * The I/O Buffer Register may be read by the Host. This implements the data output register of the Associative Memory.
- * Its contents can be copied into the I/O Register of the Common or the Mask Register. In this way it is possible to perform linked search operations, where the first associative search selects the key to the next search operation.
- * The 8 bits of the I/O Buffer Register, padded with zeroes to the left, may be loaded into the loopcounter or into the Address Processor, allowing indirect address links and loop values to be stored in the Associative Memory.

2.2.7 Status Register

The Status Register may be read at any moment by the Host. It contains the following bits of information:

BUSY	Indicates that the instruction pipeline is full. It is used in the "Host Driven Polled" mode of communication between the Host and the Control Unit. (See Section 2.4)
NONE	When this status bit is TRUE, none of the PEs has its Tag Register set to ONE. After an associative search it is often useful to know whether any word matched the search criterion.
ZAP	This bit indicates Zero-Address-Processor status.
ZLOOPC	This bit indicates Zero-Loopcounter status of the loopcounter in the Address Processor.
S1/S2	Two general purpose signals from the Control Unit. Their values are defined by a two-bit field of the control word.

They can for example be used to indicate the progression in a running microprogram.

2.3 PARALLEL PROCESSING ARRAY

2.3.1 Processing Element

The Processing Array contains 128 identical processing elements. A processing element (see Figure 2.7) consists of:

- * ALU
- * A local memory with 4096 bits
- * Four one-bit registers: T, R, C and X
- * A data selector
- * Input/Output register

The ALU is implemented in the form of a 1024 word by 4 bit PROM. Five of its ten address lines are used as data inputs, the remaining five receive the ALU function code. The four one-bit registers are connected to the outputs of the ALU and are all loaded with the ALU result at the end of each clock cycle. Input data to the ALU comes from three of the registers: the T (tag), the R (result) and the C (carry), from the Common Register and from the data selector.

Since processing is bit-serial, it is extremely important that the ALU instruction set is well suited for the kind of calculations done. Using a five-bit function code, only 32 ALU functions can be included. The implementation of the ALU in the form of a user programmable read-only memory allows its instruction set to be adapted for

different applications.

Six of the inputs to the data selector may be connected to the interconnection network, one is connected to the PE memory and the eighth input receives the value of the X (auxiliary) Register.

The PE memory is a 4096 word by one bit RAM with separate input and output lines. It receives a 12 bit address from the Address Register of the Address Processor in the Control Unit. The memory output bit is sent both to the data selector of the ALU, the I/O Register and, via a buffer, to the interconnection bus. Data input to the memory can come from the R Register or from the I/O Register (as selected by a one-bit field of the microinstruction word). Activation of the memory write input is either unconditional (WRITEALL) or tag-masked (WRITE).

The I/O Register is used to transform data from the byte format used by the Host to the bit format which is used internally in the PE. It is an 8 bit shift register with a serial input, a serial output and a two-directional parallel input/output. Data in byte format may be loaded into the shift register from the Host. Then it is written into the PE memory under microprogram control. This I/O scheme assures that the input and output data rate is not limited by LUCAS. The Host may fill the 128 I/O Registers at highest speed. Thereafter the PE memories are loaded simultaneously with one byte slice in 12 clock cycles, resulting in a data transfer rate between the I/O Registers and the PE memories of approximately 50 Mbytes/second.

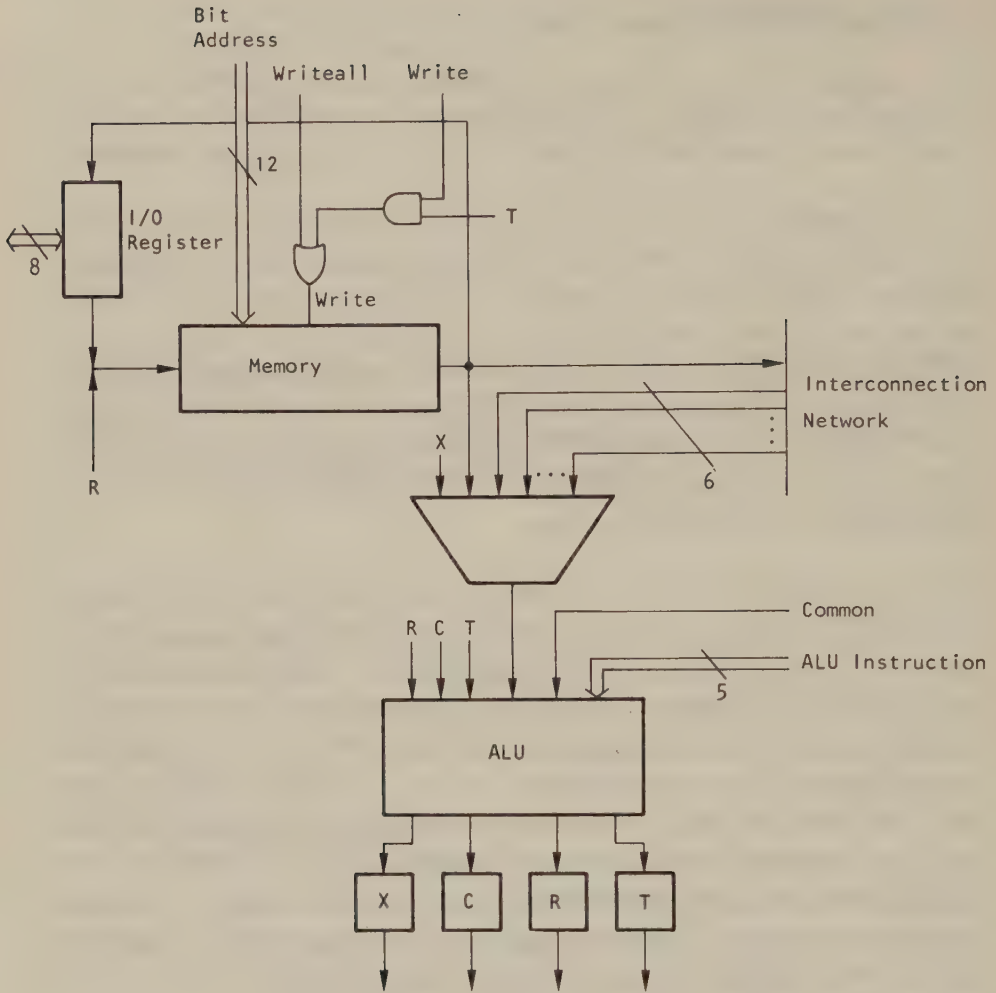


Figure 2.7 A Processing Element in LUCAS

2.3.2 Communication Between Elements

Two distinct paths for communication between PEs are present: the I/O Bus and the Interconnection Network.

2.3.2.1 I/O Bus

Communication over the I/O Bus is used when one PE, as selected by its Tag Register, is the data source and when data should be broadcasted to all or to a subset of the PEs, to the Common or to the Mask Register. One byte is moved from its source to its destination(s) as follows:

- 1) Copy the source field into the I/O registers
- 2) Select the source PE by setting its Tag Register to One
- 3) Perform a READ-SELECTED operation, which copies the I/O Register of the selected PE to the I/O Buffer Register
- 4) Perform a IOWRALL operation. The I/O Buffer Register contents is broadcasted to all the I/O Registers, including those of the Common and the Mask Registers.
- 5) If the destination is Common or Mask then perform input from the corresponding I/O Register. If the destination is one or several PEs then set the corresponding Tags and perform a tag-masked input from the I/O Registers

In applications where LUCAS acts as an associative memory, the I/O Buffer Register is used as a data register for input and output. Data retrieval is done by letting the Host read the I/O Buffer Register

after step 3 in the procedure above. Data which should be written into the Associative Memory is first placed in the I/O Buffer Register whereafter steps 4 and 5 are executed.

2.3.2.2 Interconnection Network

The interconnection network comprises a 128 bit wide bus and a data selector in each PE. The outputs of the PE memories are connected to the interconnection bus, where one line is reserved for each PE. On the input side, a strapping area on the PE boards allows the six free inputs to the data selector of a PE to be connected to any of the 128 lines on the bus. Currently four of the six connections are used.

Conceptually, communication over the interconnection network differs in several aspects from communication over the I/O Bus.

- * The source and the destination of all links are fixed and not data dependent.
- * Transport of data is done in parallel between different source-destination pairs, resulting in a permutation of the input data.
- * All the PEs participate.
- * Communication between PEs, where no direct link exists, can be obtained by multiple passes through the network.
- * Transport of data is bit-serial, not byte-serial

Assume a linear ordering of the PEs, with indices reaching from 0 to 127. The permutations currently defined on LUCAS are:

- * Rotate Down. Data from the memory of PE_i is presented

to the ALU of $PE_{(i+1) \bmod 128}$.

- * Rotate Up. Accordingly in the other direction.
- * Shuffle. Data is permuted according to the perfect shuffle described in Section 1.6.2
- * Neighbour-shuffle. This corresponds to a shuffle of the input vector, followed by a pairwise exchange.

Exploiting the possibility of tag-masked operations, the interconnection network can act as a one-stage shuffle-exchange network with individual control of the exchange elements: first all the R Registers are loaded from the shuffle inputs with the ALU instruction LRMA. In the next step, the Tag contents is used to control the pairwise exchange when the ALU instruction LRMT reloads the R Registers of certain PEs from the neighbour-shuffle inputs.

From a general point of view we can say that data communication over the interconnection network is useful when the interconnection structure is regular and independent of the data values. It has been used in matrix operations, image processing and Fast Fourier Transform. Communication over the I/O Bus is used in applications where the associative feature of LUCAS is predominant, which means that the current data values determine which PEs should communicate. It has mainly been used in database processing.

2.3.3 Select First Chain

The Select-First/Some-None chain is depicted in Figure 2.8. Each of the 16 boards containing the PEs has an input and an output for the Some/None chain. The output of a board is activated if the input is active or if any of the internal Tag Registers are set (there are eight PEs on one board). The output from the last board is stored in a register in the Control Unit and may be used as a test condition by

the Sequencer.

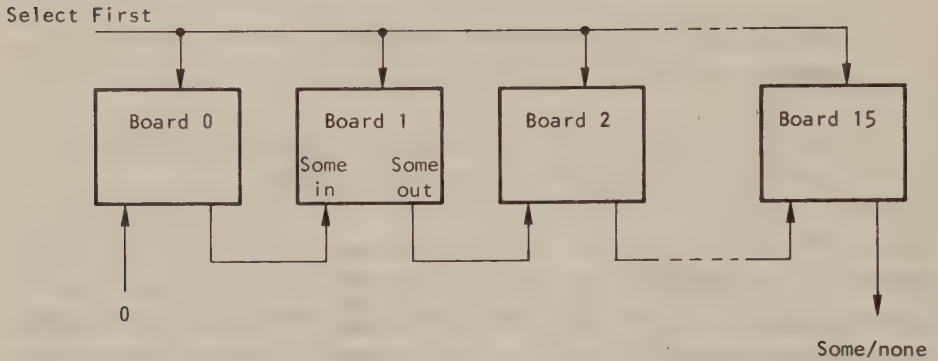


Figure 2.8 The Some/None chain

When the Select First function is activated, the effect is that all the Tag Registers which are set will be cleared, except for the one with the lowest index.

The implementation of a multiple match resolver in the form of a Select First chain, as seen in Figure 2.9, limits the maximum number of PEs in LUCAS. The longest propagation delay in this chain occurs when only the first and the last PEs have their Tags set. The time needed to clear the last Tag Register in this case is:

$$T_{cl} = (B-1) * g + (n-1) * g + s,$$

where B is the number of boards used, n the number of PEs per board, g the gate delay and s the set-up time for the clear function of the Tag Register. On LUCAS we have $B=16$, $n=8$, $g=3$ ns (TTL-S technology is used for this purpose) and $s=10$ ns, resulting in $T_{cl}=77$ ns, which gives a good margin to the 200 ns clock cycle period used.

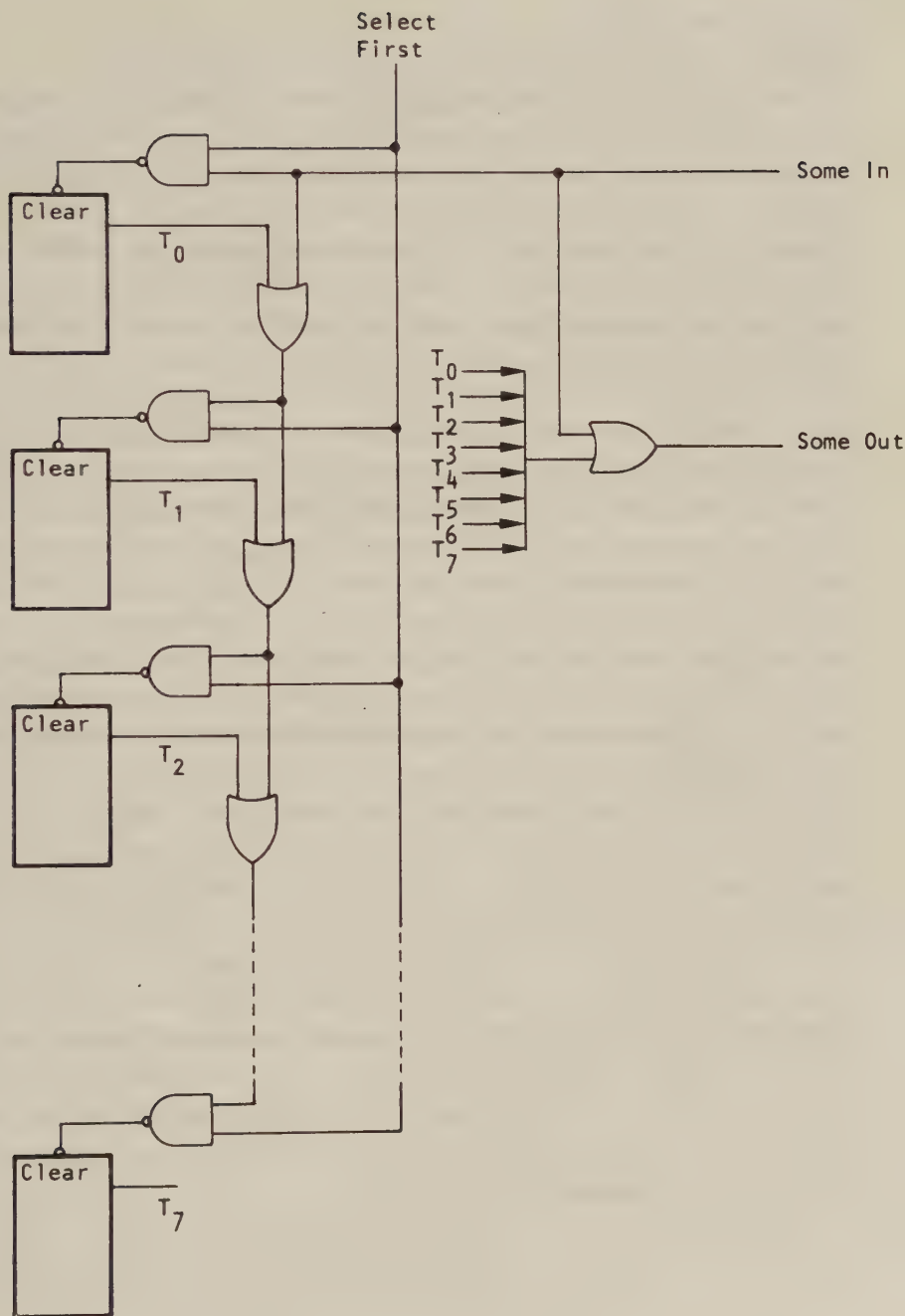


Figure 2.9 Select First and Some/None logic of one PE board

2.4 HOST COMMUNICATION

In the Control Unit the instructions are stored in a one-level instruction pipeline. When the Control Unit has started the execution of an instruction, a second instruction can be loaded into the pipeline. If a third instruction is sent, the Host is automatically brought to a wait-state and its execution is temporarily halted. When the execution of the second instruction starts in the Control Unit, the pipeline becomes empty and the new instruction is loaded, whereupon the Host resumes its execution.

This is one of the three possible modes of communication between the Host and the Control Unit. We refer to it as "Host Driven Automatic" mode, or HDA mode. It is used in applications where the computations in the Associative Array set the speed limit, i. e. in programs where the parallel operations predominate over sequential operations. It is always preferable that neither the Host nor the Control Unit waits for the other, however in a program where most of the processing is done by the PEs, instructions must be sent at highest possible rate even if this means that the Host sometimes becomes idle. In order to get an optimal throughput of instructions in both the Host and the Control Unit, the instruction scheduling is extremely important: having sent an instruction to the Control Unit, the Host should itself execute instructions during the same amount of time it takes for the Control Unit to complete its execution.

Other modes of communication are: "Host Driven Polled" mode (HDP mode) and "Control Unit Driven Interrupt" mode (CDI mode). In the first of these, the Host reads the status of the instruction pipeline and does not issue a new instruction until the Control Unit is ready to accept it. In the second mode, a Host interrupt is initiated when the instruction pipeline becomes empty.

The HDP mode is used when debugging programs, in order to prevent

the complete system from "hanging" due to a looping microprogram. Before sending an instruction, the Host checks if the instruction pipeline is free. If this is not the case, it rechecks a limited number of times and if the Control Unit is still busy, an error is reported to the user.

The CDI mode is more complex than the other two. In this case, the programs that are executed in the Host and in the Control Unit can be seen as two asynchronous concurrent processes. They are asynchronous since even though all instructions are located in the Host's memory, the Control Unit reads its own instructions by interrupting the Host and this is transparent to the program in the Host. The two concurrent processes share two common resources: the I/O registers and the Status Register in the Control Unit. To guarantee exclusive access to the common resources, software semaphores must be used.

So far, the possibilities of the CDI mode have not been fully investigated. If it will be more used, more efficient ways of synchronization and protection of common resources should be considered. The reason for including this mode, which seems more difficult to handle and involves more overhead than the HDA mode, is that once an efficient solution to the synchronization problem has been found, programs can be written without the need for the instruction scheduling mentioned earlier. As soon as instructions for the Control Unit appears, they are moved to an instruction buffer where they are fetched by the interrupt routine. In a way this instruction buffer would act as a many-leveled instruction pipeline.

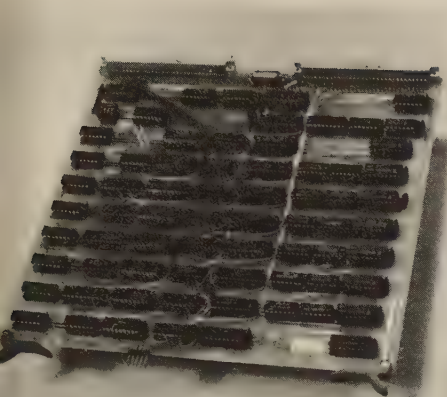
2.5 PHYSICAL IMPLEMENTATION

LUCAS is implemented mostly in Low Power TTL (TTL-LS) MSI circuits with some parts in bipolar LSI technology. This limits the speed to a maximum clock pulse frequency of 6 MHz. The use of Advanced Low Power TTL (TTL-ALS) and Emitter Coupled Logic (ECL) would allow an increase of the clock pulse frequency by a factor 3 to 4. Currently the machine runs at 5MHz speed.

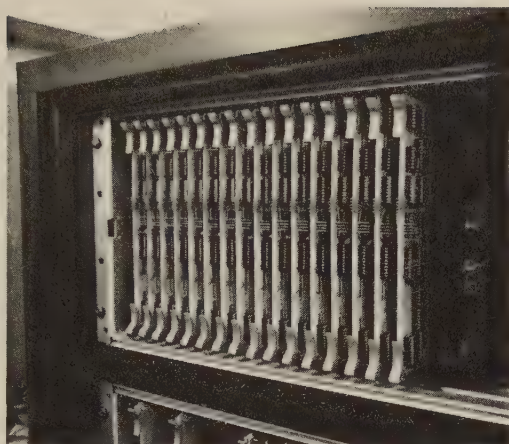
The Associative Array in LUCAS consists of 128 Processing Elements which are housed on 16 printed circuit boards. Each board holds eight PEs on an area of 220 x 230 mm. Together with buffers and decoding logic, the eight PEs use a total of 70 IC packages and consumes approximately 3 amperes, see Figure 2.10 (a). The complete Associative Array together with the interface to the Control Unit, bus terminations and cooling fans fills one 19" rack of 310 x 480 x 260mm. This is depicted in Figure 2.10 (b).

The Control Unit, is wire wrapped, and fills one board of 220 x 230 mm. The microprogram memory uses 2147 HMOS memories with 4096 words by one bit per package. The 80 memory packages plus the pipeline register fills one 220 x 230 mm board. It consumes approximately 18 amperes and has separate cooling fans.

The complete system, which includes the Associative Array, the Control Unit, the Host system, a floppy disk unit with two 8" drivers and a graphical display processor is contained in the cabinet, which is shown in Figure 2.11.



(a)



(b)

Figure 2.10 (a) Board with 8 PEs. (b) Associative Array

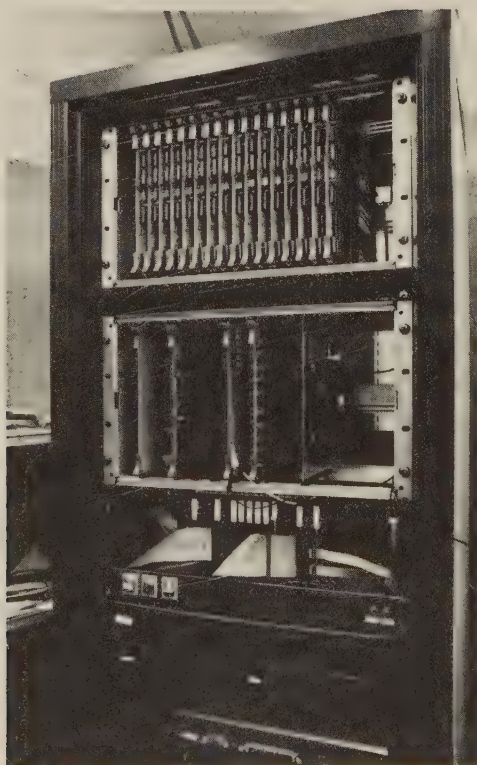


Figure 2.11 LUCAS

Chapter 3

BASIC SYSTEM SOFTWARE

3.1 INTRODUCTION

This chapter describes the programming of LUCAS and introduces the basic software support which has been developed.

Every application program on LUCAS is divided into two parts: one part which is executed sequentially in the Host and one part which executes in parallel on data located in the Associative Array. The parallel part consists of microprograms for the Control Unit of LUCAS. The sequential part, which may be written either in assembly language or in high-level language, takes care of scalar processing and external I/O. In addition to this, the Host program is also responsible for providing the instructions to the Control Unit. Input and output of data to and from the Associative Array involves both the Host and the Control Unit: the Host loads or reads the PE I/O Registers and the Control Unit executes microprograms for moving data between the PE memories and the I/O Registers.

3.2 SOFTWARE SUPPORT

LUCAS has four programming levels with different support tools (see Figure 3.1). The lowest level ("picocode") is used to define the instruction set of the PE ALUs. These are implemented as programmable read-only memories (PROM), which allows a change in the instruction set in order to tune LUCAS to a certain application. A table generator has been developed, which takes Boolean expressions of the inputs of the ALU (see Section 2.3.1) and produces the PROM contents in the form of a binary hex file. The file format is a standard format accepted by most PROM programmers. The current instruction set, which is of general purpose type and supports both arithmetic and Boolean operations, is listed in Appendix 1.

PROGRAMMING LEVEL	BASIC SUPPORT TOOL	ELABORATE SUPPORT TOOL	OTHER TOOL
HIGH-LEVEL LANGUAGE	Procedure Library	Pascal/L (not implemented)	
MACHINE CODE	Macro Assembler Library		File Handler
MICRO CODE	Micro Program Assembler	LUCAS Micro Programming Language	Microprogram Debugger
"PICO CODE"	Table Generator		

Figure 3.1 LUCAS Programming environment

3.2.1 Microprogramming

Microprogramming is supported by several tools:

- * Microprogram assembler
- * High-level like language for microprogramming
- * Microprogram debugger

The microprogram assembler was developed at an early stage of the design phase, when it proved to be an important tool for the

debugging and testing of the hardware. It is implemented in the form of a macro definition library to a commercially available macro processor (MAC from Digital Research) and runs on the Host system. For details of its use, see [Kru-23]. Details of the microinstruction format and the microprogramming instruction set is found in Appendix 1.

A language has been designed for microprogramming of LUCAS. It allows microprograms to be written on an "algorithmic level", which means that the programmer does not need to deal with low level hardware control (enabling buffers, waiting for signals to propagate through the system etc.). It uses a control structure which is similar to high-level languages (if-then-else, while etc.). A compiler, which runs on a VAX 11/780, produces microcode of approximately the same quality as hand coded microcode. The language and the compiler will be described in detail in Chapter 4 and 5 respectively.

The Microprogram Debugger, AMON, allows microprograms to run at full speed on LUCAS with a breakpoint facility. The contents of both the Microprogram Memory and of the Associative Memory may be displayed, changed, loaded from or written to disk. The program consists of several modules, which are loaded in one piece:

- * Microprogram Memory Monitor. This module allows microprograms to be loaded from disk, the contents of the microprogram memory to be displayed in symbolic form (microprogram disassembler) and the contents of the microprogram memory to be changed by using symbolic notation (microprogram line assembler).
- * Execution Monitor. This module allows the user to load the Parameter Registers and start a microprogram. A breakpoint may be inserted in which case the microprogram runs at full speed up to this point, where the execution is halted.
- * Associative Array Monitor. The contents of the PE

memories and of the Common and the Mask Registers may be displayed and updated using several different formats (hexadecimal, ASCII characters, integers or fixed point numbers). The PE registers may be inspected and an area of the Associative Array may be stored on disk or loaded from disk. The PE memory contents may be output to the graphical display processor with a varying pixel format (1 to 8 bits per pixel), resulting in a "pixel mapped" image on the TV screen of the display unit.

- * Communication Module. Allows transparent communication with the LUNET terminal switching network. The microprogram compiler, which is described in Chapter 5, runs on a VAX 11/780 which is connected to LUNET. The communication module allows microprograms to be developed on the VAX and transferred directly to the microprogram memory of LUCAS.

3.2.2 Machine Level Programming

The LUCAS assembly instruction set is a collection of standard microprograms which can be used to implement the Control Unit part of most programs. These instructions, which constitute the standard software interface to the Host, are of the form:

INSTRUCTION, P1, P2,P3, P4

INSTRUCTION is the instruction code and the P_is are the optional parameters, which serve the purpose of operand descriptors to the instruction. The meaning of the parameters is standardized as follows:

- P1 gives the location of the first source operand
- P2 gives the location of the second source operand
- P3 gives the location of the destination
- P4 specifies the length of the operand(s)

In Figure 3.2 a number of the LUCAS assembly instructions are listed.

INSTRUCTION	P1	P2	P3	P4	FUNCTION
SETTA					Set all Tags to ONE (TRUE)
SELF1					Select first TRUE Tag
LDOA	X				Load I/O Regs from source field
WRINA			X		Write I/O Regs to destination field
RSEL					Copy the I/O Reg of single selected PE to the I/O Buffer
MOVFA	X		X	X	Copy field from source to destination
MOVFT	X		X	X	Same as above but in selected PEs only
INTADDT	X	X	X	X	Add integers in selected PEs
INTMULT	X	X	X	X	Same as above but multiply
INTMAXT	X			X	Find the maximum integer in the source field of selected PEs. Deselect other PEs
INTMINT	X			X	Same as above but minimum

Figure 3.2 Example of LUCAS assembly instructions

Assembly programs are written in the assembly language for Z80 and are processed by a macro assembler on the Host. A macro library

defines constants and macros, which are used to communicate with LUCAS. The macro LUC is used to reference the LUCAS assembly instructions. A call to this macro includes one to five parameters where the first parameter specifies the instruction and the remaining four the values to be put in the Parameter Registers. The instruction codes for the LUCAS assembly instructions are predefined in the library file. The macro call has the following syntax:

```
<LUC-call> ::= LUC  <instr> {,<prm>}
<instr>      ::= SETTA | SELF1 | RSEL | LDOA | WRINA | ..
<prm>        ::= <empty> | <constant>
```

Using this macro has two advantages. First, the programs become more compact and readable. Second, the program code becomes independent of the mode of communication with the Control Unit (see Section 2.4). By specifying the value of an assembly constant in the beginning of the source program, the user selects either HDP mode (for debugging) or HDA mode to be used for the expansions of LUC.

A File Handler is defined as a collection of subroutines which can be linked to user programs in order to provide a file handling capacity for the Associative Array.

3.2.3 High-Level Language Programming

In order to use LUCAS in a high-level language environment, a set of basic procedures has been defined which constitute a high-level software interface to a Pascal system, which runs on the Host. Since the interaction between LUCAS and the Host is extremely simple, the interface which is written in assembly code, could be kept small. In the implementation which will be described in this section, the total size of the assembly code is less than 1000 bytes. The Pascal system which is used in the experimental implementation, Pascal-Z, is one of several Pascal compilers available for the current Host, Z80.

The procedures are contained in a library and define the following functions:

- * Declaration of variables which are located in the Associative Array.
- * Data movement between variables in the Host and variables in the Associative Array and also between external files and the Associative Array.
- * Arithmetical and logical operations where the operands are variables in the Associative Array.

Before we describe any of the library procedures, we will look at an example of how an application program which uses the library can be written. The program reads a datafile containing 128 8 bit signed numbers into the Associative Array. It then sorts the data in descending order and fills the Host variable TAB with the sorted values by repeatedly reading and removing the maximum value from the Associative Array.

```

1  Program Filesort(input,output);
2  const    include 'LUCCONST.PAS'
3
4  type      include 'LUCTYPE.PAS'
5            string = packed array[1..20] of char;
6            intarray = array[1..128] of integer;
7
8  var        include 'LUCVAR.PAS'
9            tab : intarray;
10           filename : string;
11           i,j : integer;
12
13  include 'LUCAS.PAS'
14
15  Procedure Sort(datafile : string;
16                var table : intarray; var k : integer);
17
18  var      A : field;
19  begin
20          lucmark; lucdeclare(A,8);
21          lucintreadf1(datafile,A);
22          k:=1; lucintmax(A);
23          while lucsome do
24          begin
25              lucselectfirst; lucrsel;
26              tab[k]:=lucintreadiobuff; k:=k+1;
27              lucremove(A);
28              lucintmax(A);
29          end;
30          lucrelease;
31  end;
32
33  begin { MAIN }
34      lucinit;
35      write('File to sort: '); readln(filename);
36      sort(filename,tab,i);
37      for j:=1 to i-1 do
38          writeln(tab[j]);
39  end.

```

Figure 3.3 Application program using the LUCAS library files

As different Pascal systems have different ways of defining libraries

and declaration of library procedures, we use an INCLUDE directive in the example. The INCLUDE directive simply instructs the compiler to insert the contents of the specified file into the program.

In order not to violate the Pascal rule governing the order of the declarations, the "library" is divided into four files. The first three files contain declarations of constants, types and variables which are used by the library procedures and which are also defined in the application program. The last file included contains the library procedures. The names of the library procedures have the prefix LUC.

The main program starts with a call to LUCINIT (line 34) which assures that no variables are allocated to the Associative Array. After reading the filename, it calls the procedure SORT. This procedure declares one local variable in the Associative Memory, namely A, which is of type FIELD. The FIELD type is defined in the library and refers to a field in the Associative Memory (see Figure 1.9).

Every procedure which declares local variables in the Associative Memory starts with a call to LUCMARK and ends with a call to LUCRELEASE (lines 20 and 30) - the meaning of these will be explained later. Then the field must be explicitly declared by invoking LUCDECLARE. Parameters to LUCDECLARE specifies the name of the field and the number of bits each element contains. On line 20 the field A is declared to be 8 bits wide.

LUCINTMAX on line 22 selects the PE containing the maximum value in the field and the following loop is repeated as long as a maximum value exists: the maximum value is put in TAB, removed from the field and a new maximum is searched. Upon return to the main program the sorted table is output on the console.

The use of the procedures LUCMARK, LUCDECLARE and LUCRELEASE allows the same rules of access and scope that exist in Pascal to apply to the field variables. The scope rule says that a variable is defined only within the block (program, procedure or function) where it is declared. It is defined in any inner block unless another

variable with the same name has been declared in the inner block. This is a static scheme. Since Pascal allows recursion, a variable can have several occurrences at the same time during the execution of a program. The dynamic access rule says that a reference to a variable should be associated with the memory location(s) which was reserved for the most recent invocation of the block where the variable is declared.

As was seen in the example, a block where a field variable is declared, includes a variable declaration of the following kind:

```
var      A : field;
```

FIELD, which is type declared in the library file LUCTYPE.PAS, is a record which serves as a descriptor of the field in the Associative Memory (see Figure 3.4).

```
type      bitadr = 0..4095;
          field = record
                        data   : bitadr;
                        size   : integer;
                        range  : bitadr;
          end;
```

DATA refers to the address of the rightmost (least significant) bit of the variable in the Associative Memory, SIZE holds the number of bits in the variable and RANGE is the address of a bitslice where each bit indicates whether the particular PE contains field data or not. This bitslice is later initialized when data is loaded into the field.

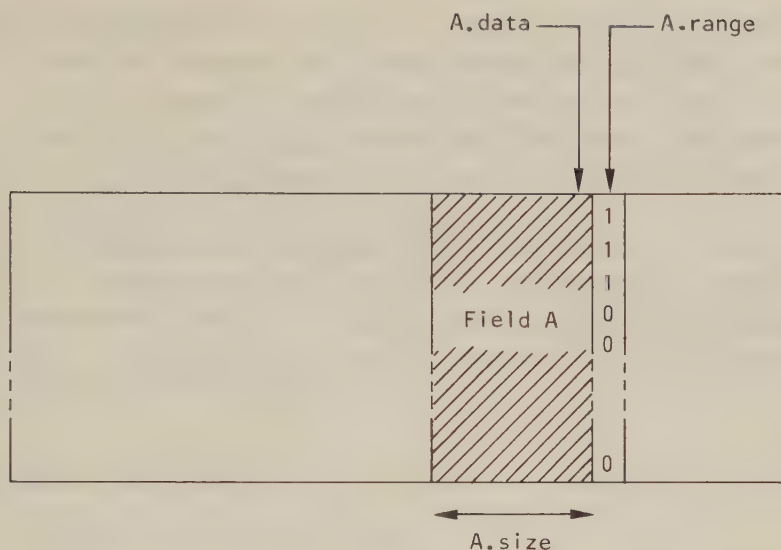


Figure 3.4 The field record is a descriptor of the field in the Associative Memory

Because the field descriptors are declared as ordinary Pascal variables, they follow the scope and access rules of Pascal.

To initialize a descriptor, a call to LUCDECLARE is necessary. LUCDECLARE allocates the variable to a free area in the Associative Array by giving values to the DATA and the RANGE components of the FIELD record.

Most (all?) block structured languages allocate data on a stack structure. The reason for this is twofold:

- * The implementation of the dynamic access rule is facilitated. The most recent occurrence of a variable is the one closest to the stack top.

- * Upon exit from a block, all local variables can be discarded and the memory space they occupied be freed by simply changing the value of the stackpointer, which indicates where the free memory area starts, to the value it had when the block was entered.

In the LUCAS procedure library, the Associative Array is treated as a stack with a stackpointer, TOS. Each time LUCDECLARE is called, the specified descriptor is initialized so as to point to the free area immediately above TOS, and TOS is changed accordingly.

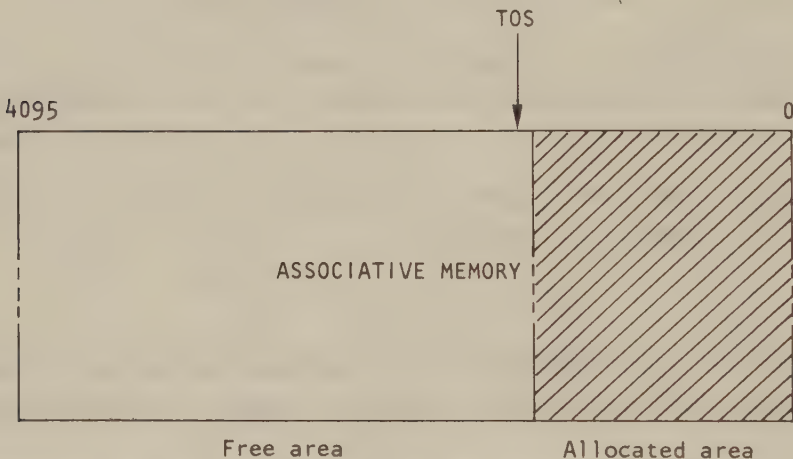


Figure 3.5 Allocation of data in the Associative Memory

Now a mechanism is needed to release the area that was reserved for the local variables, when the procedure or function returns. For this purpose a stack is defined where the TOS value is pushed. This is done by invoking LUCMARK each time a block which declares local field variables is entered. Before leaving such a block, LUCRELEASE is called which restores TOS to the value it had when the block was entered. The role of LUCINIT is to initialize TOS to zero and to

clear the TOS stack.

```

type    bitadr = 0..4095;
var     TOS : bitadr;
        stack : array[0..15] of bitadr;
        stackpointer : 0..15;

Procedure Lucinit;
begin
    TOS:=0; stackpointer:=0;
end;

Procedure Lucmark;
begin
    stack[stackpointer]:=TOS;
    stackpointer:=stackpointer+1;
end;

Procedure Lucrelease;
begin
    stackpointer:=stackpointer-1;
    TOS:=stack[stackpointer];
end;

Procedure Lucdeclare(var fname : field; fsize : integer);
begin
    with fname do
    begin
        data:=TOS+1;
        range:=TOS;
        size:=fsize;
    end;
    TOS:=TOS+fsize+1;
end;

```

Figure 3.6 Declarations and procedures defining the memory allocation scheme

Figure 3.6 shows the declarations and the procedures which implement the allocation scheme for the Associative Memory.

For data movement between standard Pascal variables and field variables, several procedures are defined. These procedures either read or write a Pascal array in the Associative Memory. The Pascal arrays may be of type Boolean, integer or char and they can have either one or two dimensions. The procedure call specifies the number of dimensions (different procedures for one and for two dimensional arrays) and the number of elements that should be moved in each dimension. The field size, which was given in the LUCDECLARE statement, specifies the number of bits in the elements. Examples of data movement operations are:

```
LUCINTREAD1(A,TAB,100)
```

```
LUCCHARWRITE2(TEXT,B,128,30)
```

The LUCINTREAD1 procedure reads a one-dimensional integer array from the Associative Memory to a Pascal array. The first parameter, A, gives the source in the Associative Memory, the second parameter, TAB, is the destination array and the third parameter refers to the number of elements that will be read. The field size of A determines the number of bits which are used for each element. If this number is smaller than the number of bits used for integer variables in the Pascal system, the data will be right justified with the sign bit copied in the remaining bits. In case the field size is larger than the number of bits in the Pascal integers, this results in a run time error.

The LUCCHARWRITE2 procedure copies a two dimensional array of characters to the field specified in the second parameter. The last two parameters give the number of elements in each dimension.

Other procedures exist for reading and writing files to the Associative Memory, to read and write scalars in the Common

Register, the Mask Register and in the I/O Buffer Register.

Procedures for specifying arithmetic and logical operations on field variables correspond directly to the instructions in the LUCAS assembly instruction set (see Figure 3.2). They have the same names as the assembly instructions - preceeded by the prefix LUC - and the parameters correspond to the parameters of the assembly instructions, except that the operand length is implicitly given by the field size in the declaration, e. g. LUCMOVFA(A,B) which copies the contents of the field A to the field B including the range bitslice which indicates in what PEs the field A is defined.

In conclusion to the use of a procedure library can be said that it allows programs written in a high-level language to interact with LUCAS without too much effort needed for the implementation. Once the basic structure, such as the memory allocation scheme, has been decided, new procedures can easily be added when new microprograms are written. What mainly reduces the usefulness of the approach is the overhead involved. Memory allocation is done at run time instead of being handled by the compiler, every instruction that is sent to LUCAS results in Pascal procedures being called. Also, error checking is minimal and operations on field variables can not be described in the form of expressions.

Part 3. LUCAS Microprogramming Language

Chapter 4 LANGUAGE DEFINITION

4.1 INTRODUCTION

LUCAS is used in applications where its capabilities of processing structured data in parallel can be utilized to obtain a significant increase in processing speed over a standard minicomputer. The bit-serial nature of the computations performed by LUCAS is both convenient and efficient for processing low-precision data, which is the case in for example image processing and database management. However, it is extremely sensitive to inefficiencies in the implementation of the bit-serial algorithms. The innermost loops are often very tight, typically 2-5 machine cycles, and one or two unnecessary instructions in the loops have a tremendous impact on the processing speed. This means that most users will be forced to write some parts of their programs in microcode.

Disadvantages of programming in microcode are well known, and not specific for LUCAS. The programmer must be very well acquainted with the underlying architecture of the machine and take into consideration all peculiarities with for example timing and pipelining.

If microprogramming is horizontal (as in LUCAS) it is possible to let microoperations, which control different parts of the machine overlap, and be executed simultaneously. Deciding which operations could overlap requires a detailed knowledge of the internal workings of the computer and optimal or near optimal solutions are very hard and time consuming to find. On the other hand, it is important to exploit this possible parallelism.

With an increasing number of microprogrammable computers appearing, the interest in microprogram development tools is growing rapidly. The use of high-level like languages for microprogramming is appealing. A language which incorporates typical high-level qualities, such as block structure, procedures and structured control statements of type if-then-else for selection and while for repetition, will support modern programming methods, which facilitate both development, maintenance and documentation of programs. Another important aspect is that the language compiler could be responsible for the tedious task of forming the complete microinstructions with optimal (or near optimal) overlapping between the microoperations. Such a language is not automatically machine independent but it is obvious that a unified approach to the design of languages of this kind could be adopted.

Current research in the area of microprogramming techniques have two directions: the design of suitable programming languages and the development of methods for optimizing the microcode.

The request for highly efficient microcode influences the definition of languages. Most designers agree that the firm demand for machine independent languages, which prevails in the area of high-level languages, is not applicable to microprogramming. Ramamoorthy and Tsuchiya formulate their point of view in [Ram-1]:

"The desirable properties of a high-level microprogramming language must be a compromise between machine dependency, ease of detection and representing explicit and implicit parallelism, and the innate 'naturalness' required of all programming languages to establish effective man-machine communications."

It deserves to be mentioned that the essential point in the argument for machine independence of high-level languages, is that programs should be transportable between different computers. It is reasonable to assume that the need for this in the context of microprogramming is very small. Therefore it is less important if two microprogramming languages differ in their details. What is important is that they use similar basic constructs and that a unified machine independent methodology for their implementation is developed with machine independent techniques for compilation and optimization.

The MPG system [Bab-1] is an example of a machine independent approach to high-level microprogramming. A program written in the language (MPGL) has two parts: a machine description (MDS) and an algorithm description (ADS). The MDS is used to initialize the compiler and to prepare it for translation of the ADS part, which consists of machine specific algorithms.

A similar technique has been proposed by DeWitt [DeW-2]. His language includes "extension statements" and "extension operators" which allow the definition of new data types and operations. The core language includes a high-level like control structure, integers and primitive assignment statements. The result is similar to the MPG system: a program consists of a declaration part where the extensions are defined and an algorithm part which is machine dependent.

In [Das-2] Dasgupta discusses the meaning of the term high-level microprogramming language. He concludes that a microprogramming language could be quite primitive and still be called high-level if it has certain machine- and implementation-independent characteristics, such as structured control constructs and a possibility to "describe and name, arbitrarily, microprogrammable data objects".

In the same paper, he presents his approach to machine independent microprogramming, which takes the form of a generic family of languages, called a schema. The schema includes all the possible constructs and data types that can be used when the schema is instantiated into a language for a particular machine. This

instantiation defines the specific properties of the machine in the form of declarations of the data objects which are available (registers, memories, stacks, flip-flops etc.) and specifications of the possible operations on these data objects. The schema includes several high-level control structures (if-then-else, case, while, repeat), a primitive data type, which is the bit, and five structured data types: sequence of bit, array, stack, record and associative memory.

The microprogramming language which will be presented in this chapter was defined with the following design objectives:

- 1) It should have a machine independent frame work defining a general program structure which resembles that of high-level languages.
- 2) The machine dependent characteristics which are added should allow a change of the PE instruction set without the need to re-define the language and with minimal changes needed in the compiler.
- 3) The semantic of the language should be close enough to the machine to allow an efficient implementation, where the code produced is of the same quality as hand written microcode

The resulting language has a structure where microprograms, together with subroutines they use, are grouped together in modules and where variables and constants may be declared either on the module level or locally in the microprograms or the subroutines. The control constructs are of high-level type. The language allows microprograms to be written on an algorithmic level without the need to deal with low-level hardware control. In fact, no explicit programming of the Control Unit needs to be done, which means that the programmer deals with a much less complex architecture.

4.2 MICROPROGRAMMER'S VIEW OF LUCAS

This section describes the architecture of LUCAS as seen by the microprogrammer when using the microprogramming language. According to this view, LUCAS consists of an array of 128 processing elements (PEs), a Common Register, a Mask Register and an I/O Buffer Register (see Figure 4.1). Processing of data is done in a bit-serial fashion by the PEs.

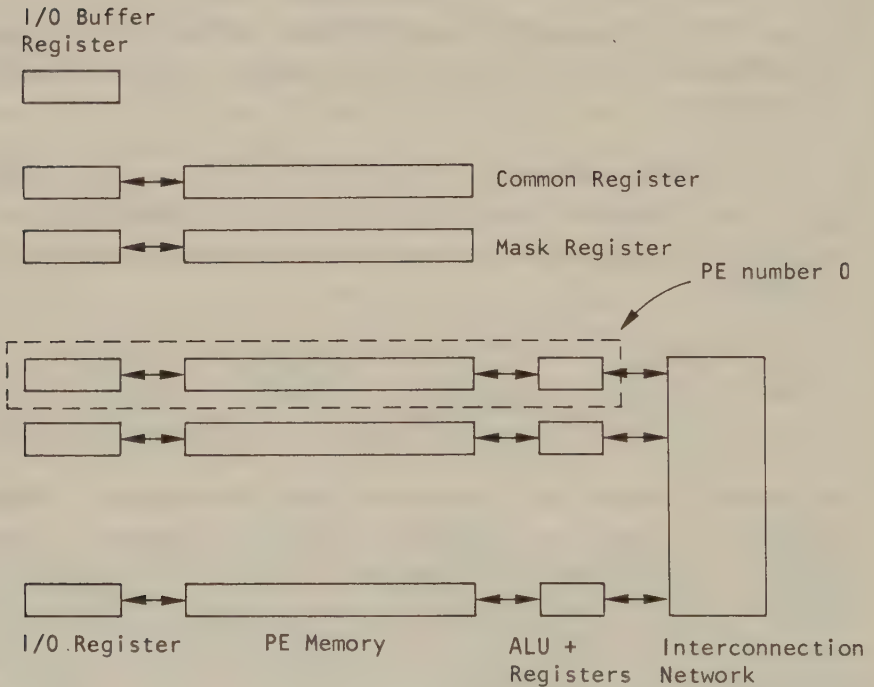


Figure 4.1 The Microprogrammer's view of LUCAS

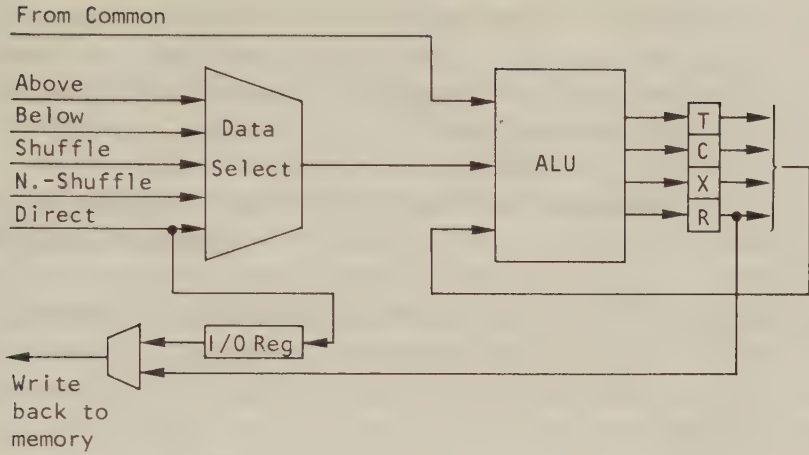


Figure 4.2 Processing Element (PE) in LUCAS

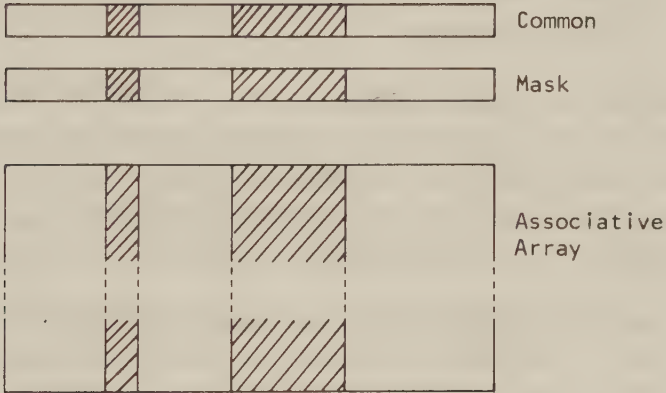


Figure 4.3 Bit-slice and field in the Associative Array

A PE (see Figure 4.2) comprises an ALU, four one-bit registers, an eight-bit input/output register and a 4096 bit memory. One of the registers, the Tag (T), is used to "select" the PE. When the Tag has

the value ONE, the PE will execute every instruction it receives. If the Tag has the value ZERO, only instructions which are destined for all the PEs will be executed and other instructions will be interpreted as no-ops by the PE.

In LUCAS all data paths are one bit wide. This means that only one bit of data is accessible at a time from the PE memories, the Common, the Mask and the I/O Registers. This is commonly referred to as a bit-slice of data. Every operation on data which is more than one bit wide, must be executed as a sequence of operations. Data of this kind, which is composed of a number of bit-slices, is called a field (see Figure 4.3).

Instructions to the PEs are of the following kind:

- * Input/Output. Write the I/O Register output value into the memory, shift the memory output value into the I/O Register. Similar instructions are used for the Common and Mask I/O Registers.
- * Register operations within the PE. Instructions executed in the ALU. Operands are the four one-bit registers and the memory output bit, which is connected to the ALU via the "direct" input of the multiplexer (see Figure 4.2). The result of the operation is stored in the four registers.
- * Register operations with data from other PEs. The same type of operations as above, but data from some other PE's memory is connected through the multiplexer according to the communication network which is defined on LUCAS.
- * Register operations with data from Common. Again the same type of operations, leaving the result in the registers, but now the output from the Common Register is used as an additional operand.
- * Memory write. The value of the R Register is written

into the PE memory.

- * I/O Buffer Register operations. The value in the I/O Register of a selected PE (assuming exclusive selection) can be copied into the I/O Buffer Register and then be broadcasted to all the I/O Registers, including those of Common and Mask.
- * Operations on the Tag Registers. Several of the operation types described above are affected by and affect the Tags. In addition to these, there are instructions to set all Tags (to ONE) and to reset all Tags except one.

4.3 INTRODUCTION TO THE LANGUAGE

4.3.1 Program Structure

The entire program text presented for compilation constitutes a compilation unit. The syntactical construct that represents the basic compilation unit, is the module. A module starts with a module heading and terminates with the keyword endmod. The body of the module has two parts, a declaration part and a submodule part, of which any may be empty.

An entity declared in the declaration part of a module is known, or visible within the module and may be referenced from any point following its declaration. Declared entities are of the following types:

- constants
- variables
- subroutines

The submodule part can take two forms. It can consist of one or

several new modules, nested within the basic module, which is then called the global module. Or it can be composed of a number of microprograms.

The microprograms serve the purpose of interface to the environment and are the only entities visible outside the compilation unit. A microprogram defines a machine level instruction for LUCAS and is invoked from the Host computer when it sends an instruction to LUCAS. Once a microprogram has started, it runs to completion without being interrupted. It terminates its execution with a microcode branch to a predefined location and LUCAS informs the Host that a new instructions may be sent. The only place in the compilation unit where a microprogram is visible is in its own body - it can be explicitly aborted, but it can not reinvoke itself.

The concept of a subroutine differs from that of a microprogram in that the subroutine is visible in the module where it has been declared, but not outside the compilation unit. A subroutine may be invoked from a microprogram, from another subroutine or recursively from itself. Recursion is allowed, but should be avoided since the stack used for subroutine nesting is limited to five(!) levels.

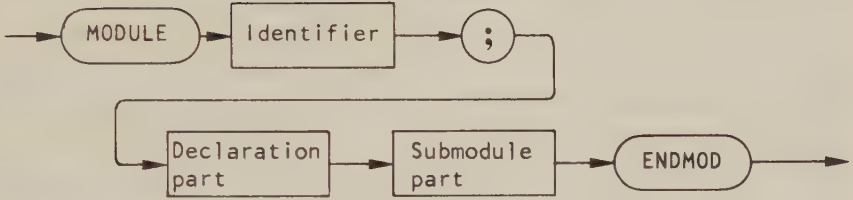
Both microprograms and subroutines can have parameters and may declare local constants and variables. Figure 4.4 gives the syntax diagrams for the general structure of a compilation unit.

The scope of a declaration is defined as the part of the program text over which the declaration has an effect, i. e. where the name of the declared entity is known. Figure 4.5 illustrates the general scope rules of the language.

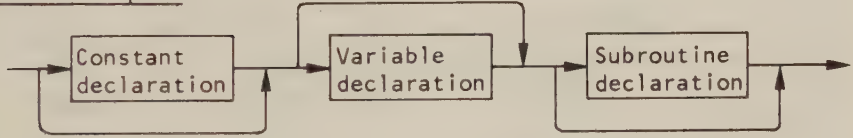
Compilation Unit



Module



Declaration part



Submodule part

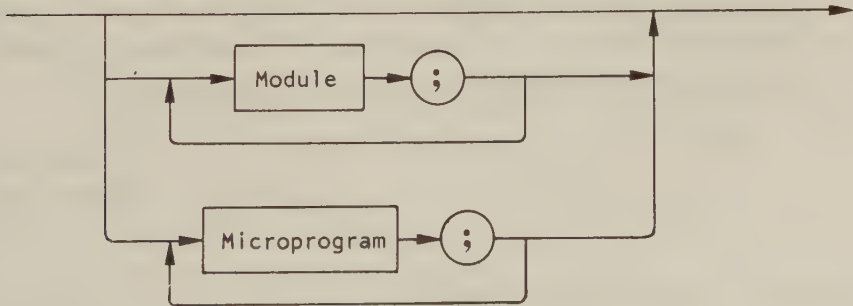


Figure 4.4 The general structure of a compilation unit

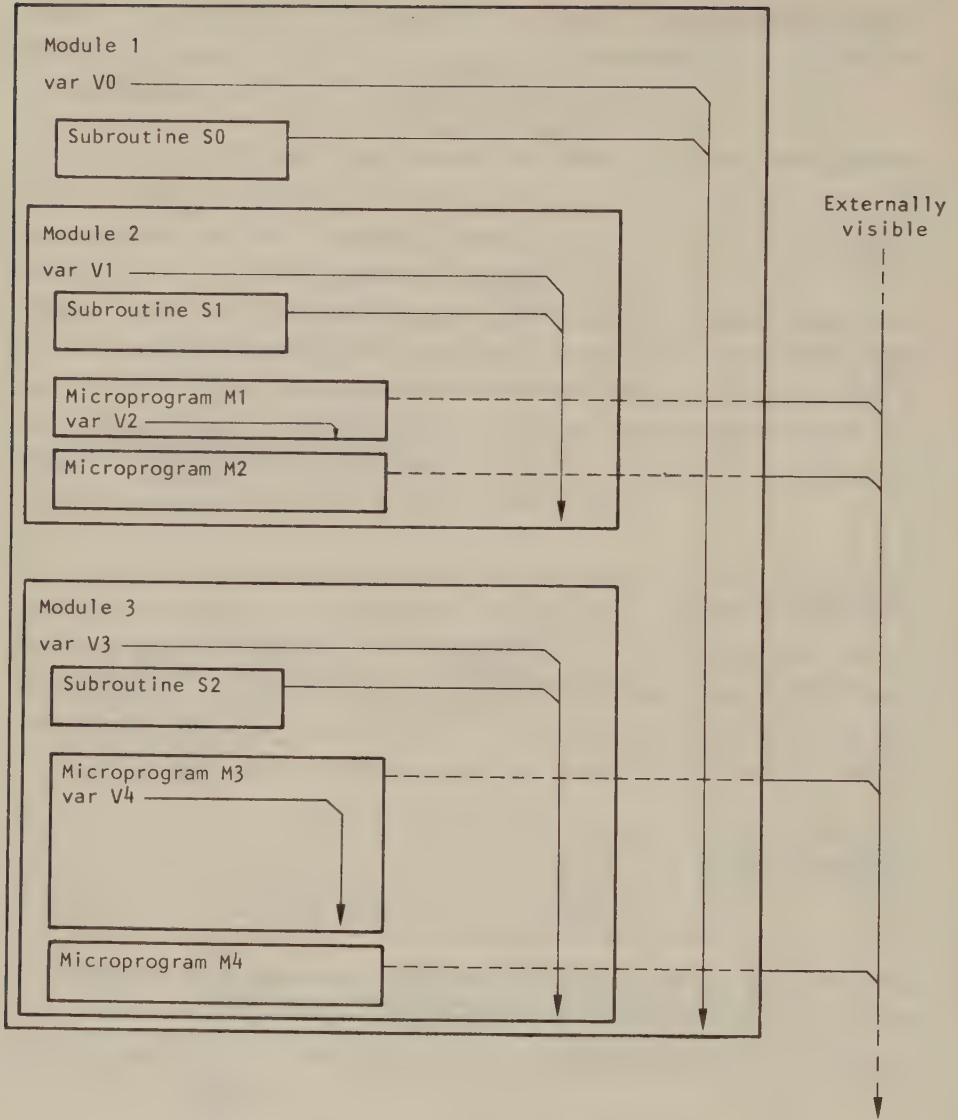


Figure 4.5 Scope rules in nested modules

4.3.2 Notation

A slightly modified form of the Backus-Naur form (BNF) will be used to represent the syntax of the language. A BNF description contains: terminals, nonterminals and metasymbols.

A terminal is a basic symbol in the language with a fixed interpretation. Terminals are separators, operators and keywords:

Separators

; , () := :

Operators

+ - = ≠ (which may be written as <>)

Keywords

begin call const do else end endmod
exit if iterate microprogram module
repeat subroutine then times
until var while

In BNF and in the program examples, all keywords are underlined.

Nonterminals are special symbols, metalinguistic variables, used in the BNF. Their meaning is to help provide a hierarchical structure of the language description. In accordance with the BNF convention, we will use the metasymbols < > to enclose nonterminals.

Metasymbols are used to denote the productions, or rewriting rules, of the underlying grammar.

Metasymbols

< > ::= | { } }^x

::= should be interpreted as "means" or "can be rewritten as". | should be interpreted as "or". The curly brackets, { }, are introduced

in order to denote repetition of the enclosed symbols zero or more times (cf. [Jen-1]). (This allows a BNF description of the language without recursive rules.) When the terminating curly bracket is indexed, such as }^x, the meaning is that the enclosed symbols may be repeated at most x times.

<empty> denotes the empty string.

4.3.3 Lexical Rules

The vocabulary of the language is composed of the terminals described above in conjunction with user defined strings, which are either identifiers or numbers. Programs are written in a free format with the only rule that adjacent identifiers, keywords and numbers must be separated by at least one delimiter. Delimiters are the terminal separators and operators along with the special characters space, horizontal tabulation and carriage return.

An identifier is a name which is introduced by the programmer to denote a constant, a variable, a module, a subroutine or a microprogram. The identifier consists of a letter followed by an arbitrary number of letters or digits.

A number consists of a digit, which is followed by an arbitrary number of digits. Hexadecimal notation can be chosen by means of a compiler directive, in which case the opening digit also may be followed by the letters A, B, C, D, E and F.

Any string starting with a '{', followed by an arbitrary number of characters upto and including a '}', is treated as a comment.

4.4 LANGUAGE ELEMENTS

4.4.1 Constants

A constant is an identifier which is associated with an integer value at the time of its declaration. This identifier may later be used in the program text in place of the integer value it represents. The constant declaration has the following form:

```
<constant declaration> ::=
    const <identifier>=<sign><constant> {;<identifier>=<sign><constant>;}
```

```
<sign> ::= <empty> | -
```

```
<constant> ::= <number> | <constant identifier>
```

where the number is in the range 0 to 4095,

The association between the identifier and the value is valid throughout the module, unless the same identifier is redeclared in a module, subroutine or microprogram which is nested within the module where it was originally declared.

```
const  c1 = 100;
        c2 = 256;
```

4.4.2 Variables, Assignments

A variable declaration introduces the identifiers which are used to denote the variables in the program text. Declaration of variables is either implicit or explicit. Parameters to subroutines and microprograms (see Sections 4.4.3 and 4.4.4) are treated as locally declared variables within the body of the subroutine (microprogram). The format of the explicit variable declaration is:

```
<variable declaration> ::=
    var <identifier>{,<identifier>;}
```

A variable may be assigned values in the range 0 to 4095. The same rules of scope applies to variables as to constants.

```
var v1,v2,v3;
```

A variable is used in the following contexts:

- * Pointer to data in the Associative Array. The current value of the variable is used to indicate a bit-slice in the Associative Array.
- * Test variable in control constructs. The value of the variable is tested in the condition part of the control statements.
- * Assignments. A variable can be assigned a new value.
- * Stack Operations. A variable can be pushed onto or popped from a predefined stack.

A value assigned to a variable in one microprogram is valid in any other microprogram of the module until either the variable is re-assigned or a microprogram of some other module is executed.

In the current implementation of the language, a maximum of sixteen variables may be visible at any point in the compilation unit. (This is due to the allocation scheme for the variables which are located in the registers of the Address Processor.) However, this restriction does not normally cause any problem if local variables are used whenever possible.

Variables are given values in assignments. The assignment statement has the following general form:

$$\langle \text{assignment} \rangle ::=$$

```
<variable1>:<sign><constant> | <variable2> |  
    <variable1><operator><constant> |  
    <variable1><operator><variable2>
```

$$\langle \text{operator} \rangle ::= + \mid -$$

This means that a variable can be assigned:

- * the value of a constant or number, optionally negated.
- * the value of any variable (including itself).
- * its current value plus or minus a constant or a number.
- * its current value plus or minus the value of any variable (including itself).

```
v1:=256;  
v1:=v4;  
v1:=v1+256;  
v1:=v1+v2;  
v1:=v1+v1;
```

Variables can be pushed onto or popped from a predefined stack by use of the standard procedures SPUSH and SPOP. However, the same stack is implicitly used to maintain certain variables at subroutine invocation (see Section 4.4.3). This means that a variable which has previously been pushed onto the stack, must be popped in the same routine (microprogram or subroutine) where it was pushed. Any subroutine calls between the push and the pop operation does not affect the value of the pushed variable.

```

    SPUSH(v1);
    ...
    SPOP(v1);

```

4.4.3 Subroutines

A subroutine declaration associates an identifier (a name) with a part of the program text in a module. The declaration consists of a subroutine heading, local declarations and an executable body.

```

<subroutine declaration> ::=
    <subroutine heading><local declaration part><statement part>

```

```

<subroutine heading> ::=
    subroutine <identifier>; |
    subroutine <identifier>(<formal subroutine parameter list>);

```

```

<formal subroutine parameter list> ::=
    <identifier>{,<identifier>}

```

```

<local declaration part> ::=
    <empty> | <constant declaration> | <variable declaration> |
    <constant declaration> <variable declaration>

```

```

<statement part> ::=
    begin <statement list> end

```

```

<statement list> ::=
    <statement>{,<statement>}

```

The parameter passing convention is "call-by-value", which means that the identifiers which are (implicitly) declared in the formal parameter list of the subroutine heading are conceptually equivalent to local

variables of the subroutine. They are initiated from the calling program part, whereas any additional local variables have undefined values when the execution of the subroutine starts.

The subroutine may be invoked from any executable part of the module where it is declared (unless the identifier denoting the subroutine has been redeclared). At the time of invocation, the actual parameters, which are specified in the subroutine call, defines the initial values of the formal parameters of the subroutine declaration. Actual parameters can be either variables or constants.

<subroutine call> ::=

call <identifier> | call <identifier><<actual parameter list>>

<actual parameter list> ::=

<actual parameter>{,<actual parameter>}

<actual parameter> ::=

<variable identifier> | <sign><constant>

Since a subroutine never is nested in another subroutine or a microprogram, all communication of values must either be carried out by means of subroutine parameters or by assigning values to variables declared at the module level. The "call-by-value" convention does not allow a subroutine to return a value by assignment to a formal parameter.

4.4.4 Microprograms

The microprogram concept has a strong correspondence to that of the subroutine and the declaration of a microprogram is similar to the subroutine declaration, only the heading is different.

```

<microprogram heading> ::=
    microprogram <identifier>; |
    microprogram <identifier><<formal microprogram parameter list>>;

```

```

<formal microprogram parameter list> ::=
    <microprogram parameter>{,<microprogram parameter>}3

```

```

<microprogram parameter> ::=
    <empty> | <identifier>

```

The formal parameter list includes a maximum of four parameters, separated by ','. A parameter which occurs in the list is implicitly declared as a local variable to the microprogram.

The difference between a subroutine and a microprogram lies in the invocation procedure. A subroutine is called from inside the module where it is declared, whereas no mechanism is defined in the language for calling a microprogram. Instead, microprograms are invoked from programs at another level: the assembly level of the Host.

Parameters passing is of "call-by-value" type; the Host initiates the parameter variables before the microprogram is started. The hardware interface between LUCAS and the Host allows at most four parameters to be passed through the Parameter Registers of the Control Unit. As seen in the formal definition of the parameter list, a parameter may be left blank, which denotes that no value is passed through the corresponding Parameter Register.

```
Microprogram M1(p1,p2);
```

means that parameters are passed through Parameter Registers 1 and 2.

```
Microprogram M2(p1,,,p2);
```

means that parameters are passed through Parameter Registers 1 and 4.

The microprogram terminates its execution by branching to a predefined routine, OPFETCH, which effects the transfer of the next instruction from the Host and starts the corresponding microprogram.

Communication of values between two microprograms is possible via the variables which are declared on the module level. A typical situation where this is used is when a microprogram needs more than four parameters. In this case, the microprogram is put in a module together with an auxiliary microprogram which assigns its parameters to variables declared on the module level, as follows:

```
Module M1;
var par1,par2,par3,par4;

  Microprogram Loadparam(p1,p2,p3,p4);
    begin
      par1:=p1; par2:=p2; par3:=p3; par4:=p4
    end;

  Microprogram Mic(param5,param6,param7,param8);
    begin
      ... /When Mic starts par1 - par8 are defined
    end;
endmod.
```

4.4.5 Statements I - Program Flow Control

4.4.5.1 General

The body of a subroutine or a microprogram contains executable statements which are grouped in a statement list. A statement list is a (possible empty) list of statements separated by semicolons.

Statements denote actions which are performed in some part of LUCAS. Two basic groups of statements are defined: those that specify operations on data in the Associative Array and those that control the execution flow. Statements can be of the form "compound statements" in which case a list of statements, preceded by the keyword begin and followed by the keyword end, replaces a single statement. Statements may also be empty.

<statement> ::=

<empty> | <single statement> | begin <statement list> end

<statementlist> ::=

<statement>{;<statement>}

In previous sections we have already come upon two kinds of statements: the assignment statement and the subroutine call. These will not be further discussed.

4.4.5.2 Conditions

Most of the constructs used for program flow control are conditional in that they specify two possible ways to proceed in the execution. The outcome is chosen depending on the value of the condition part of the construct. A condition can take the values "true" and "false".

<condition> ::=

<variable><test><variable> | <variable><test>0 | TRUE | FALSE |
SOME | NONE | ZMASK(<address>) | NZMASK(<address>)

<test> ::= = | <>

<address> ::= <variable> | <constant>

The first two conditions test if two variables have the same value and if a variable has the value zero, respectively. TRUE and FALSE are predefined conditions with the values "true" and "false". SOME has the value "true" if at least one PE has its Tag Register set. NONE is the complement of SOME. ZMASK(address) is "true" if the Mask Register has the value zero in position address. NZMASK is the complement of ZMASK.

4.4.5.3 If-Then-Else

The if-then-else construct is used to select one of two possible paths in the program flow. In an abbreviated form of the construct, the if-statement, the else part is left empty.

<if-then-else statement> ::=

if <condition> then <statement> else <statement>

<if statement> ::=

if <condition> then <statement>

Note that the if-then-else construct is one single statement and that no semicolon must precede the else part.

It is possible to nest several if-then-else statements, in which case the following simple rule must be respected: each else introduces the

alternative statement to the most recently encountered then, when reading the complete statement from left to right. This means that in the statement:

```
if <condition1> then if <condition2> then S1
      else S2 else S3
```

S2 is the alternative statement to S1, and S3 is the alternative statement to:

```
if <condition2> then S1 else S2
```

4.4.5.4 While

The language has three loop constructs for specifying repetition of statements. One is the while statement, which has the following form:

```
<while statement> ::=
  while <condition> do <statement>
```

Before each repetition of the statement part, the condition is evaluated. If it is "true", the statement will be executed. If it is "false", the loop terminates and the execution continues with the statement following the while construct.

4.4.5.5 Repeat

The repeat statement is similar to the while statement, the difference being that the condition which is used to control the repetition, is tested at the end of the loop, not at the beginning as in the while

statement.

<repeat statement> ::=

repeat <statement list> until <condition>

A minor difference is that the construct specifies repetition of a list of statements rather than of one single statement. The keyword-pair repeat-until serves the additional purpose of statement brackets, replacing a begin-end pair.

4.4.5.6 Iterate

In many cases, the number of times a loop should be repeated is known in advance. This is especially common in bit-serial processing, where the basic loops have to be iterated a fixed number of times, depending on the precision of the operands. Using a while statement, such a basic loop has the following form:

```
b:=noofbits;
while b>0 do
  begin
    ...
    b:=b-1
  end;
```

We note that the loop control variable, b in the example above, is accessible within the loop, where it for example may be used as a pointer to data in the Associative Array. However, very often the loop control variable needs not be accessed in the loop since it is merely used to control the iteration. This kind of loop can be very efficiently implemented on LUCAS (by the use of special-purpose loopcounters).

A loop construct of this kind is defined in the language:

```

<iterate statement> ::=
    iterate <value> times <statement>

```

```

<value> ::= <variable identifier> | <constant>

```

The value specified must be greater than or equal to 1.

4.4.5.7 Exit

An exit statement specifies a structured termination of a loop, a subroutine or a microprogram. Within a loop the exit statement will cause an immediate termination of the loop and execution will continue with the first statement following the loop structure, regardless of the value of the termination condition specified in the loop heading. Exit from a subroutine means that the control is transferred to the calling program. The effect of an exit from a microprogram is a branch to the OPFETCH microprogram.

```

iterate b times
begin
    CMCT(direct,fielda);
    if NONE then exit;
    fielda:=fielda+1;
end;

```

The example shows the innermost loop of a parallel search operation, where data from a field in the Associative Array is compared to the contents of the Common Register. (The CMCT instruction compares the Common Register contents to the contents of the PE memory at bit address "fielda".) Normally the loop is executed b times, where b is the word length, but with the use of the exit statement, the execution of the loop is terminated when all Tags are false.

In the case of nested loops, that is when one loop structure is part of the body of another loop, a simple exit causes termination of the smallest loop enclosing the exit statement. However, any enclosing loop, subroutine or microprogram may be terminated by specifying its name in the exit statement. In the case of subroutines and microprograms, the name used is the name given in the declaration of the subroutine/microprogram. A loop structure may be given a local name by using a label (identifier followed by ':') immediately preceeding the loop heading.

```

LOOPA: while b<>0 do
    begin
        ...
    repeat
        ...
        if b=0 then exit(LOOPA);
        ...
    until c=0;
    ...
end;

```

A name specified in an exit statement must be a local name, i. e. defined in the subroutine or the microprogram where the exit statement occurs. By definition we include the name of the subroutine or microprogram in the local names.

4.4.6 Statements II - Array Operations

4.4.6.1 PE Instructions

The PE instructions embrace operations performed on the registers and on the memory in the Processing Elements. The instructions are of three kinds:

Without parameters. These instructions use the PE registers both as operands and to store the result.

With one parameter. These instructions use the PE registers as operands but store the result in the PE memory. The parameter specifies the PE memory address.

With two parameters. These are instructions where one of the operands comes from the interconnection network. The first parameter gives the PE memory address of the source bit. The second parameter specifies the permutation of data over the network.

The PE instruction set may be altered by reprogramming the ALU PROMs. The current instruction set is given in Appendix 2. The following are examples of PE instructions:

LTRA	Load T from R in All PEs
LTRT	Load T from R Tagmasked (in selected PEs)
WRR(adr)	Write R into the PE memories in All PEs at bit address "adr"
LRMA(adr,ABOVE)	Load R from Multiplexer in All PEs. ABOVE specifies that the data should come from the memory of the PE immediately above in the Associative Array.

4.4.6.2 Input and Output

Input and output of data is physically handled by the I/O Registers, which are loaded either from the Host or by microcode. The language includes instructions for this purpose.

Output of data is accomplished by shifting the I/O Registers while specifying the bit-slice address of data to be output.

```
RSHIFT(address)
```

Normally one byte of data is output at a time, starting with the least significant bit.

```

iterate 8 times
begin
    RSHIFT(v);
    v:=v+1
end;

```

Input of data is handled in a similar fashion.

```

WRIA(address)
    or
WRIT(address)

```

The first instruction causes the I/O Register output bit to be written into the memory of all PEs. WRIT is the tag-masked correspondence to WRIA. None of these instructions actually shifts the I/O Register. This must be carried out by means of the SHIFT instruction, which causes all the I/O Registers to be shifted right one step.

```

iterate 8 times
begin
    WRIT(v); SHIFT;
    v:=v+1;
end;

```

Note that input and output of data may be performed in one single loop. This will result in a more efficient code than if two separate loops are used.

```

iterate 8 times
begin
    WRIT(inadr);
    RSHIFT(outadr); {send data to the
                     { I/O Register and shift}
    inadr:=inadr+1; outadr:=outadr+1;
end;

```

4.4.6.3 Common, Mask and I/O Buffer Operations

The Common Register and the Mask Register are both implemented in the form of 4096 bit random access memories, similar to the PE memories. They receive the same bit address as the PEs. The output from the Common Register may be used as an operand in certain PE instructions and the Mask output is used in conditions (see Section 4.4.5.2).

Both the Common and the Mask Register communicate with a corresponding I/O Register. These I/O Registers are either loaded from the Host or by microcode, exactly as the PE I/O Registers. Data is output to the I/O Registers with the same instruction that outputs data to the PE I/O Registers: RSHIFT(address). Note that, since the data in the Common and Mask Registers is static, output of data is

normally not meaningful.

Input of data is accomplished with the instructions

```

        WRCOM(address)
        and
        WRMASK(address)

```

A write instruction does not shift the I/O Registers. This must be specified separately.

```

iterate 8 times
begin
    WRCOM(v); SHIFT;
    v:=v+1
end;

```

The SHIFT instruction causes all the I/O Registers (including those of the PEs) to be shifted one step to the right.

The I/O Buffer Register provides a flexible communication link between PEs, between PEs and the Common and the Mask Registers, and also between PEs and the Host. It can be loaded either with the I/O Register contents of a selected PE or from the Host. Data in the I/O Buffer Register may be broadcasted to all the I/O Registers and then, by using selective input from these, be written into its destination.

```

LDIOBS      LOAD I/O BUFFER SELECTED. Load the I/O Buffer
             Register from the I/O Register of a selected
             PE (must be uniquely selected).

IOBWRALL     Copy the I/O Buffer contents to all the
             I/O Registers

```

Example Move one byte at location 'source' in the tag-selected PE to location 'destination' in every PE where the R Register is ONE.

```

iterate 8 times
begin
    RSHIFT(source);
    source:=source+1;
end;
SELF;    {Just to be sure}
LDIOBS; IOBWRALL;
LTRA;
iterate 8 times
begin
    WRIT(destination); SHIFT;
    destination:=destination+1;
end;

```

4.5 PROGRAM EXAMPLES

The first example shows a module with routines for 2's complement multiplication of integer fields in the Associative Array. The module contains two subroutines which implement addition and subtraction of fields of arbitrary length.

```

{*****}
{
{           Module for field multiplication
{
{
{ MULFA  A,B,C,length  / A*B -> C  in all PEs
{
{*****}

```

```
Module FieldMult;
```

```

Subroutine AddFieldsT(S1,S2,D,L);
begin
    iterate L times
    begin
        LRMA(S1,DIRECT); ADMA(S2,DIRECT); WRRT(D);
        S1:=S1+1; S2:=S2+1; D:=D+1;
    end
end;
```

```

Subroutine SubFieldsT(S1,S2,D,L);
begin
    iterate L times
    begin
        LRMA(S1,DIRECT); SUMA(S2,DIRECT); WRRT(D);
        S1:=S1+1; S2:=S2+1; D:=D+1;
    end
end;
```

```

Microprogram MULFT(Multiplicand,Multiplier,Destination,Length);
var D,M,K;
begin
    LTMA(Multiplier,DIRECT); Multiplier:=Multiplier+1;
    M:=Multiplicand; D:=Destination;
    { First iteration needs no addition since part. prod.= 0 }
    iterate Length times
    begin
        LRMA(M,DIRECT); ANDTRA; WRRRA(D);
        M:=M+1; D:=D+1;
    end;
    WRRRA(D); D:=D+1; { double sign bits }

    { Clear rest of part. prod. field }
    K:=Length-2;

```

```

iterate K times
begin
    WRRR(D); D:=D+1;
end;

{ Multiplication loop. Use double sign bits }
D:=Destination+1; Length:=Length+1;
iterate K times
begin
    LTMA(Multiplier,DIRECT); Multiplier:=Multiplier+1;
    call AddFieldsT(D,Multiplicand,D,Length); D:=D+1;
end;

LTMA(Multiplier,DIRECT);
call SubFieldsT(D,Multiplicand,D,Length);
end;

endmod.

```

The next example is from [Sve-3]. The microprogram OUTPER is used to find the outer perimeter of objects in a binary image which is stored in the Associative Array. The columns of the image matrix are stored as bit-slices (binary image) and each row is located in the memory of one PE. Starting in one point at the edge of the image which is part of the background, picture elements which belong to the background are "marked". When the complete background has been marked, the markers are moved to the picture elements of the objects' perimeters.

```

{*****}
{      "Outer Perimeter of Objects"
{
{OUTPER (source, marker, dest, width)
{
{Propagates bits in the marker field over points
{where source field is zero. Finally, the result
{is used to mark the outer perimeters of the objects
{*****}

```

Module OUTPERmic;

```

Microprogram OUTPER(sim,mim,dim,iw);      {source image, marker image,..}
                                           {... dest image, image width}
const ss=OFEO;                             {scan status}
var w,s,m,d;
begin
  while true do
    begin
      cra; wrra(ss);                       {clear scanstatus}
      w:=iw; s:=sim; m:=mim; sett;

      while SOME do
        begin                               {spread in rightmost column}
          lrma(m,above); orrma(m,below); ltra;
          andtmia(s,direct); andtmia(m,direct); lrt; wrwt(m);
        end;
        w:=w-1; s:=s+1;

        while w<>0 do                       {*** leftscan ***}
          begin
            lrma(m,direct); orrma(m,above); orrma(m,below); m:=m+1;
            orrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
            andtmia(m,direct); lrt; wrwt(m); orrma(ss,direct); wrra(ss);

            while SOME do
              begin
                lrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
                andtmia(m,direct); lrt; wrwt(m); orrma(ss,direct); wrra(ss);
              end;

              w:=w-1; s:=s+1;
            end;{while w<>0 do}

            ltma(ss,direct);

```

```

cca; if NONE then exit;           {if no change in the entire leftscan}
                                   {then finished spreading}
cra; wrra(ss);                    {clear scanstatus}
w:=iw; s:=s-1; sett;

while SOME do                      {spread in leftmost column}
begin
  lrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
  andtmia(m,direct); lrta; wrwt(m);
end;
w:=w-1; s:=s-1;

while w<>0 do                      {*** rightscan ***}
begin
  lrma(m,direct); orrma(m,above); orrma(m,below); m:=m-1;
  orrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
  andtmia(m,direct); lrta; wrwt(m); orrma(ss,direct); wrra(ss);

  while SOME do
  begin
    lrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
    andtmia(m,direct); lrta; wrwt(m); orrma(ss,direct); wrra(ss);
  end;

  w:=w-1; s:=s-1;
end;{while w<>0 do}

ltma(ss,direct);
cca; if NONE then exit;           {if no change in the whole rightscan}
                                   {then finish spreading}
end; {while true}

w:=iw; s:=sim; m:=mim; d:=dim;

while w<>0 do                      {Put border in dest image}
begin
  ltma(s,direct); lrta;
  if SOME then
  begin
    andtmia(m,above); andtmia(m,below); m:=m-1;
    andtmia(m,direct); m:=m+2; andtmia(m,direct); cot;
    wrwt(d);
  end
  else m:=m+1;
  w:=w-1; s:=s+1; d:=d+1;
end

```



```

    end;
end;    {microprogram OUTPER}

endmod.

```

4.6 CONCLUSIONS

A language for microprogramming of LUCAS has been defined. The language is machine dependent in the sense that it defines primitives for operation on the basic elements of LUCAS, such as the PE registers, the Common and the Mask Registers. However, it is built on a general base which defines an overall program structure as well as control constructs similar to those of high-level languages. The iterate-statement which has been introduced in the language has a very natural application to bit-serial processing.

The module concept is a structured approach to the use of global variables. It has three important consequences:

- * It reduces the number of variables that are visible at some point in the program.
- * Subroutines and microprograms which are used together may be put in the same logical unit.
- * A microprogram which needs more than four parameters may be formed as a module containing several separate microprograms which are invoked in sequence.

Experience from microprogramming in microprogram assembly code has shown that the part of LUCAS which is the most difficult to program is the Control Unit, and especially the Address Processor which will become a bottleneck if not properly handled. The use of local and global variables in the language, arithmetic expressions on variables

and automatic handling of parameter passing to subroutines makes the Address Processor disappear from the programmer's view. This, together with the high-level like control constructs of the language, have resulted in a very powerful tool for microprogram development. The readability of programs is also very good, which simplifies the documentation.

Chapter 5

LANGUAGE IMPLEMENTATION

5.1 INTRODUCTION

This chapter describes the implementation of a compiler for the microprogramming language which was defined in Chapter 4.

The translation process for a microprogramming language resembles that of an ordinary high level language in many aspects and standard compiler techniques may be employed in several parts of the compiler. The main difference lies in the code generation scheme. The horizontal microinstruction format, which is used in many modern computers, provides a potential for parallelism between the functional parts of the machine which must be utilized in order to assure maximum performance. Recent research has produced a number of strategies for recognizing the parallelism in microprograms and several algorithms have been proposed for microcode optimization. The term compaction is most often used in this context, since the algorithms aim at a reduction in the size of the code without any claims of obtaining an optimal solution.

The compiler which has been developed for the LUCAS microprogramming language [Fer-3] uses a compaction algorithm based on the First-Come First-Served algorithm as proposed by Dasgupta and Tartar [Das-1]. This algorithm has been modified both to give a

better performance and also to allow microoperations which execute during more than one machine cycle.

The compiler has five layers, or phases, as shown in Figure 5.1. The first three of these constitute the first compilation pass, i. e. when the compiler reads the source code file. The code improvement layer performs several passes over the intermediate code, which is scanned back and forth during this phase. The last phase is basically a two-pass assembler which produces the final microcode.

The presentation of the compiler is in no way complete and should be regarded as intentional rather than an exact description of the implementation. In the cases where code is given, it is often simplified in order to focus on the most important aspects. The compiler is written in Pascal and code examples are given in Pascal-like notation. The size of the compiler is approximately 3000 lines of source code.

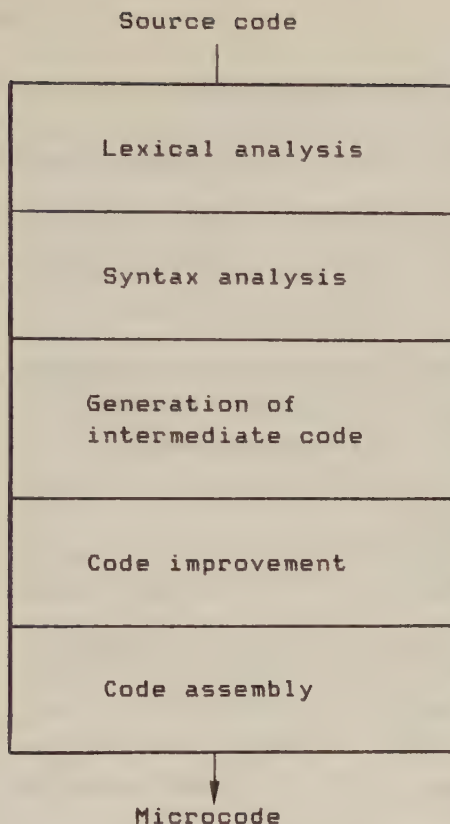


Figure 5.1 Data flow in the compiler

5.2 LEXICAL ANALYSIS

Lexical analysis is done by the procedure Scan, which transforms the source code from a string of characters to a string of symbols and which removes any comments which appear in the program text. It recognizes for example that a certain sequence of characters represents a keyword or an operation of the language and produces a corresponding symbol in its output. The following symbols are

produced by Scan:

Keyword symbols	For example BEGINS, ENDS, WHILES, DOS.
Operator symbols	These are PLUS, MINUS, NEQ, EQU, BECOMES
Punction symbols	For example PERIOD, COMMA, SEMICOLON, COLON
Identifier symbols and	
Constant symbols	These have associated values which give the name and the actual value respectively.

For example, the source statement

```
while p1 <> 100 do p1:=p1+1;
```

is translated by Scan to the following sequence of symbols:

```
WHILES
IDENTIFIER      name='p1'
NEQ
CONSTANT        value=100
DOS
IDENTIFIER      name='p1'
BECOMES
IDENTIFIER      name='p1'
PLUS
CONSTANT        value=1
SEMICOLON
```

The implementation of the Scan routine in the form of a finite automaton is straight-forward as described in the standard literature on compilers, for example [Gri-1, Aho-1].

5.3 PARSER

The parser reads the symbol stream produced by Scan. It analyses of the syntactic structure of the source program text and is responsible for the following tasks:

- * Building an (implicit) parse tree which defines the syntax of the program being compiled.
- * Checking that the program is syntactically correct. If not, an error is reported and an error recovery routine tries to re-synchronize the parser with the input stream in order that parsing can continue.
- * Calling the appropriate semantic routines when a syntactic construct has been recognized (syntax directed translation). The semantic routines are of two kinds: either they are used to enter information in the symbol table concerning variables, constants and subroutines, or they are code generating routines which produce the intermediate code.

5.3.1 Syntax Analysis

The analysis of the source code is of top-down type. This means that the parse tree which corresponds to the syntax of the compiled program is created from its root with nodes developed in preorder (leftmost derivation). For example, to expand a node with children $c_1, c_2, \dots, c_n$, first the subtree which has c_1 as its root is expanded then the subtree which has c_2 as its root and so on.

Legal expansions of the nodes are given by the grammar of the

language. In our case the grammar is of type LL(1), which means that by looking at the next symbol in the input stream, the parser can always decide which grammar rule should be used to expand the current node.

The implementation of the parser is of type recursive descent. It uses a number of recursive procedures to recognize the input. Each procedure corresponds directly to one production rule of the grammar. Guided by the current input symbol, a procedure analyses the input string with respect to the production rule. By doing this it may or it may not call any of the other procedures and it may or it may not call the semantic routines.

The following example shows how a production rule of the grammar is represented in one of the parser procedures. This example also shows the recursive nature of the parser.

```
<statement> ::=
    ... | begin <statement>{;<statement>}<end | ...
```

```
Procedure Statement;
begin
    ...
    if INSYMBOL = BEGINS then
    begin
        Scan;
        repeat Statement
        until INSYMBOL <> SEMICOLON;
        if INSYMBOL = ENDS then Scan else ErrorReport;
    end else
        ...
end;
```

The last symbol produced by Scan is held in the global variable INSYMBOL which is of enumeration type. If INSYMBOL has the value

BEGINS the parser knows that the syntax of the input should be according to the production rule in the example. First Scan is invoked in order to produce the next symbol. Then the procedure Statement is invoked recursively and this is repeated until INSYMBOL has the value SEMICOLON upon return from Statement. (Note that every procedure of the parser is responsible for leaving in INSYMBOL the first symbol which follows upon the part it has processed.) If the syntax of the compiled program is correct, INSYMBOL should now have the value ENDS and Scan is invoked to produce the next symbol.

In a recursive descent compiler it is not necessary to explicitly build the parse tree of the compiled program if the semantic routines are called during the parsing process. At any moment, the location in the parse tree of the node which is being expanded, is given by the state of the parser, i. e. the dynamic procedure invocation sequence which has led to the execution of the current procedure. By returning from this procedure, the parent node will be revisited and by continuing the execution at this level, new children of the parent node may be expanded. In this way the branches of the parse tree are built, semantic routines are invoked and the branches are cut off when no longer needed.

5.3.2 Error Recovery

When a syntax error appears in the input this is discovered in the parser by the fact that no production rule can be applied to the current input symbol. When this happens we expect the following reaction from the compiler:

- * A message which gives all possible details of the error should be produced.
- * The parser, which is now out of phase with the input stream, must "recover" so that the analysis can

proceede. It is desirable that the compiler detects as many errors as possible during one compilation. A compiler which aborts when the first error is detected is unacceptable.

- * No possible error in the compiled program must cause the compiler to go into a loop or to crash.
- * No single error in the input must cause an abundance of error messages.

Errors are reported in a listfile which is maintained by Scan. The error message, which appears in the position of the listfile where the error was discovered, is produced by the current parse procedure. It specifies the nature of the error. For example "BEGIN expected" or "Identifier or Constant expected".

The basic error recovery scheme which is used in the compiler meets the two last requirements above and to some extent also the second one. It is based on a scheme proposed by Turner [Tur-1] and has the advantage of being extremely simple to implement, since the recovery is independent of the current state of the parser.

To check the correctness of the input, the parser has two routines: MustBeIn and ScanIf. MustBeIn matches the current input symbol against a set of possible symbols, reports an error if the input symbol is not in this set and performs recovery as described below. ScanIf checks the current input symbol against the expected symbol. If they agree it calls Scan to get the next symbol. If not, it reports the error and performs error recovery.

When a first error is encountered by MustBeIn or ScanIf they return to the parser as if an expected symbol was present in the input and the syntax analysis continues until the next call to any of the procedures. If the error consisted in a missing symbol, the parser has now recovered and no further action is needed. Any other kind of error will result in a second error situation. The action taken by the routines in this case is to skip the input until the expected

symbol appears and then return to the parser. The implementation of MustBeIn and ScanIf is shown in Figure 5.2.

```

Procedure MustBeIn(GOALSET : SYMBOLSETTYPE; ERRNO : integer);
begin
    if RECOVER then
        begin
            while not (INSYMBOL in GOALSET) do Scan;
            RECOVER:=false;
        end else
            if not (INSYMBOL in GOALSET) then
                begin
                    ReportError(ERRNO);
                    RECOVER:=true;
                end;
            end;
end;

Procedure ScanIf(GOAL : SYMBOLTYPE; ERRNO : integer);
begin
    if RECOVER then
        begin
            while INSYMBOL <> GOAL do Scan;
            RECOVER:=false;
            Scan;
        end else
            if INSYMBOL = GOAL then Scan
            else
                begin
                    ReportError(ERRNO);
                    RECOVER:=true;
                end;
            end;
end;

```

Figure 5.2 Error recovery in the parser. RECOVER is a global Boolean variable which initially is assigned the value false.

Our experience from using this method for error recovery is that it

is quite sufficient for an experimental compiler. Errors which consist of single missing or replaced symbols are recovered efficiently. Problems occur when an error leads the parser to take the wrong action and to start develop an erraneous branch of the parse tree. In this case the parser never "admits" that it is on the wrong track but continues to develop the branch. This may lead to a situation where the expected symbol is not found until much later in the input and where a large part of the source program is skipped during recovery.

5.3.3 Semantic Routines

The semantic routines which are called by the parser are used either to insert new entries in the symbol table or to generate intermediate code.

The symbol table maintains information about declared constants, variables, subroutines and labels. Entries are held in the symbol table while the block in which they are declared in is parsed. New entries are added at the end of the table. When an identifier is encountered in the statemenet part of a microprogram or a subroutine, the symbol table is searched starting with the last declared entry. This assures that the most recent declaration of an identifier is used. Information about declared microprograms is stored in a separate table, which is never consulted during the compilation.

The intermediate code consists of a list of pseudo-microinstructions (PMIs). A PMI is similar to a microinstruction but with only a limited number of its fields defined. It controls the smallest meaningful activity in some part of the machine and consists of a microoperation (MO) together with its possible parameters (which also are microoperations). For example:

CJP ZAP 100

is a PMI which controls the Sequencer of LUCAS. It consists of the

MO CJP (Conditional Jump) which is an instruction to the Sequencer. CJP has two parameters: the MO ZAP (Zero Address Processor) which controls the Testmultiplexer and 100 which defines the value of the Sequencer's data field.

The following semantic procedures are defined for the generation of intermediate code:

- | | |
|----------|--|
| Gen | The single parameter of this procedure is the name of an MO. The procedure generates a new PMI containing this MO. |
| Join | Join is similar to Gen but places the MO in the last generated PMI. |
| Datajoin | This procedure has two parameters: the name of a microinstruction field and an integer value. The referenced field of the last generated PMI is assigned the integer value. |
| Insert | This procedure is used to insert a PMI in the list of PMIs already created. It has two parameters: a pointer in the list and the name of an MO which will be put in the generated PMI. Subsequent calls to Join and Datajoin will operate on the inserted PMI. |
| Chain | Chain is used when code for pipelined (= multicycle) operations is generated. During the code improvement phase the MOs are moved according to certain rules. Calling Chain assures that the last generated PMI and the next one produced will keep their consecutive order during code improvement. |

The PMI in the example above (CJP ZAP 100) is generated by the following sequence of semantic actions:

```

Gen(CJP);
Join(ZAP);
Datajoin(Seqdata,100);

```

To maintain labels in the code, four procedures are defined together with a global integer variable, NEXTLABNO.

- Newlabelin** Causes the next generated PMI to get a label. The label number is given by NEXTLABNO, which is then incremented. This procedure is used to define a new label, which has not yet been referenced.
- Newlabelout** Defines the Sequencer's branch target field in the last generated PMI to be a label. The label number is given by NEXTLABNO, which is then incremented. This procedure is used for forward references where the label location is unknown.
- Oldlabelin** Similar to 'Newlabelin', but the label number is provided as a parameter. This procedure is used to declare a label which has been introduced in a forward reference by 'Newlabelout'.
- Oldlabelout** Similar to 'Newlabelout', but the label number is provided as a parameter. This procedure is used in backward references where the label has been declared by 'Newlabelin'.

All variables of the compiled program are allocated in the sixteen internal registers of the Address Processor. Allocation is done when the variable is entered in the symbol table and it is thereafter never reallocated. This simple solution is possible since all the registers have exactly the same function.

When a subroutine is called, the compiler checks for overlapping in the allocation of registers between the calling and the called

procedure. If such an overlap exists, the Address Processor stack is used to store the overlapping registers of the calling program.

A variable which is declared on the module level or in a microprogram is always allocated to the free register which has the lowest index, while a variable in a subroutine is allocated to the free register which has the highest index. This assures a minimum of overlap for unnested subroutine calls.

The question may be raised if it is sufficient to have sixteen variables available at the source code level. In practice this has never caused any problems so far, but this is a weak point in the implementation and the register allocation scheme may be changed in the future. Another reason to do this is that nested subroutine calls always result in the saving of registers, as the local variables of two subroutines are guaranteed to overlap.

5.3.4 Translation to Intermediate Code

The intermediate code, which is generated by the semantic actions of the compiler, is a symbolic form of the microcode and would, if processed by a microprogram assembler, produce executable but inefficient microcode.

Assignments are translated into MOs which control the Address Processor. Conditions result in MOs for the Testmultiplexer. Control statements are translated to MOs for the Sequencer. PE instructions, finally, to MOs for the PEs and in the case where they have parameters, also to MOs for the Address Processor.

Figure 5.3 shows the semantic actions for different types of assignment statements. In the first example, the statement $X:=Y$ is in the input. This is translated by calling the code generating routines as follows: first Gen(LDBA) generates a new PMI which contains the microoperation LDBA for the Address Processor (see Figure 2.6). In the example above we assume that the variable X is allocated to

register i and the variable Y to register j . (The parser finds the allocation by consulting the symbol table.) Next $\text{Datajoin}(\text{RB}, i)$ and $\text{Datajoin}(\text{RA}, j)$ puts the numeric values of i and j in the RB and the RA fields of the last generated PMI.

Source Code	Semantic Actions

$X := Y$	$\text{Gen}(\text{LDBA})$ $\text{Datajoin}(\text{RB}, i)$ $\text{Datajoin}(\text{RA}, j)$
$X := 37$	$\text{Gen}(\text{LDBD})$ $\text{Datajoin}(\text{RB}, i)$ $\text{Datajoin}(\text{Adrprocddata}, 37)$
$X := X + 37$	$\text{Gen}(\text{ADBD})$ $\text{Datajoin}(\text{RB}, i)$ $\text{Datajoin}(\text{Adrprocddata}, 37)$

Figure 5.3 Semantic actions from assignments

Source Code	Semantic Actions
SOME	Gen(SOME)
X=0	Gen(TBZ) Datajoin(RB,i) Chain Gen(ZAP)
X<>Y	Gen(TAEQB) Datajoin(RB,i) Datajoin(RA,j) Chain Gen(NZAP)

Figure 5.4 Semantic actions from conditions

In Figure 5.4 the semantic actions for different conditions are shown. The first example shows how the condition SOME results in Gen(SOME) which generates a MO for the Testmultiplexer. The second condition in the figure tests the value of a variable. Gen(TBZ) and Datajoin(RB,i), with the assumption that X is allocated in register i, produces the first PMI. Chain assures that the next PMI will follow directly after the current PMI in the final microcode, i. e. that they will execute in consecutive machine cycles. Gen(ZAP) generates the second PMI which will operate on the Testmultiplexer.

Source Code	Semantic Actions
-------------	------------------

LRTA	Gen(LRTA)
WRR(X)	Gen(TBZ) Datajoin(RB,i) Chain Gen(WRALL)
LRMA(X,direct)	Gen(TBZ) Datajoin(RB,i) Chain Gen(LRMA) Join(DIRECT)

Figure 5.5 Semantic actions from PE instructions

Figure 5.5 gives examples of how PE instructions are translated. The meaning of the semantic actions should be clear from previous examples.

The parser routine which is responsible for translating the if-then-else construct uses the two other parser routines: Condition and Statement. Figure 5.6 illustrates at what point in the parsing process these routines are called.

Symbol parsed	Other parser routine called	Semantic Actions

IF		
...	Condition	Actions in Condition Join(ComplementTest) Join(CJP) i:=NEXTLABNO Newlabelout
THEN		
...	Statement	Actions in Statement
ELSE		* Gen(CJP) * j:=NEXTLABNO * Newlabelout Oldlabelin(i)
...	* Statement	* Actions in Statement * Oldlabelin(j)

Figure 5.6 Translation of the if-then-else construct

During parsing of the condition, code is generated as described in the example above. When the condition has been processed, Join(ComplementTest) complements the generated test condition and Join(CJP) and Newlabelout generates a conditional jump over the first statement-part. The compiler saves the number of the generated label in the variable i. This is used in Oldlabelin(i), where the label is inserted in the code. Actions marked by '*' in the figure are taken only if the else-part is non-empty.

Figure 5.7 shows the translation of the while-construct. A corresponding diagram may be specified for the repeat-until statement.

Symbol parsed	Other parser routine called	Semantic Actions

WHILE		i:=NEXTLABNO Newlabelin
...	Condition	Actions in Condition Join(ComplementTest) Join(CJP) j:=NEXTLABNO Newlabelout
DO		
...	Statement	Actions in Statement Gen(CJP) Oldlabelout(i) Oldlabelin(j)

Figure 5.7 Translation of the while-construct

As concerns the generation of intermediate code for the control constructs, the iterate-statement is the most intricate. Depending on the context, the loop variable may be located in either the internal counter of the Sequencer (SEQLC), the loop counter which is part of the Address Processor (APLC) or in one of the internal registers of the Address Processor.

The most efficient solution is to use SEQLC, since this allows the special loop instruction RPCT which tests, decrements and branches in one single cycle. However, as it is not possible to copy a variable from a register in the Address Processor to SEQLC, this can be used only when the iteration value is a constant. This solution is therefore chosen in the innermost loops where the iteration value is a constant.

APLC is the second best solution, since it allows overlapping between decrementation of the counter and operations in the Address Processor. This solution is chosen whenever it is not possible to use SEQLC, either because the iteration value is a variable or because SEQLC is used by an inner loop.

When it is not possible to use either of the above solutions, the loop counter is stored in a temporarily allocated register of the Address Processor. This solution is the least efficient of the three since the Address Processor, which often is a critical resource, now must use one machine cycle of the loop to decrement the loop variable.

As we want to translate an iterate construct in one single pass, it is clear that the generation of code must be deferred until the internal statement has been processed as this may contain nested iterate-statements. To keep track of how the loop counters are used, the procedure Statement has two Parameters, USESEQLC and USEAPLC, which upon return indicate if any of SEQLC or APLC is utilized within the statement. Figure 5.8 shows a part of the parser procedure which analyses and generates code for the iterate-statement. Only the first part of the procedure, where iterations with constant iteration value is processed, is shown.

Procedure Iterate;

{This procedure is called when INSymb=ITERATES}

```
var  REG : integer; {temporary register in Adr Proc}
     P   : codepointer; {points to first PMI in the loop}
     VAL : integer; {iteration value if constant}
     USESEQLC, USEAPLC : Boolean;
     I   : integer; {number of loop start label}
```

begin

```
  REG:=Allocate; {get temporary register}
  Scan; MustBeIn([IDENTIFIER,CONSTANT,NUMBER],21);
  if INSymb.KIND in [CONSTANT,NUMBER] then
  begin
    VAL:=INSymb.VALUE;
    Scan; ScanIf(TIMES,20);
```

```

P:=CurrentPMI_Pointer;
I:=NEXTLABNO; Newlabelin;
USESEQLC:=false; USEAPLC:=false;
{ process the statement inside the loop:}
Statement(USESEQLC,USEAPLC);

if not USESEQLC then
begin
    {at the end of the loop:}
    Gen(RPCT); Oldlabelout(I);

    {set up loop:}
    Insert(P,LDCT); Datajoin(Sequencerdata,VAL);
end else

if not USEAPLC then
begin
    {at the end of the loop:}
    Gen(DECLC);
    Gen(CJP); Join(NZLC); Oldlabelout(I);

    {set up loop:}
    Insert(P,LDLC); Datajoin(Adrprocddata,VAL);
end else

begin
    {at the end of the loop:}
    Gen(DECB); Datajoin(RB,REG);
    Gen(TBZ); Datajoin(RB,REG); Chain;
    Gen(CJP); Join(NZAP); Oldlabelout(I);

    {set up loop:}
    Insert(P,LDBD); Datajoin(RB,REG);
    Datajoin(Adrprocddata,VAL);
end
end else
...

```

Figure 5.8 Translation of the iterate-statement for constant iteration values

5.4 INTERMEDIATE CODE

The intermediate code is a doubly linked list of pseudo-microinstructions (PMIs). This structure has been chosen to allow convenient insertion and deletion of the elements and also to allow scanning of the list in both directions. A PMI is generated during translation by the semantic routine Gen, which also defines the principal MO of the PMI. The parameters are generated by the semantic routines Join and Datajoin. The final code which is presented to the microcode assembler (the last phase of the compiler) consists of a list of microinstructions (MIs). A MI is formed in the code improvement phase by merging PMIs according to certain rules as will be discussed in Section 5.5.

The PMI is represented by an nine-tuple: $PMI = \langle L, M, I, O, U, T, F, B, J \rangle$ where the elements are:

- L - Label. The PMI may have been assigned a label by the semantic routines Newlabelin or Oldlabelin.
- M - Microoperation set. This is the set of MOs that are defined in the PMI.
- I - Input resource set. This set contains all the resources whose contents are used by the PMI as input.
- O - Output resource set. Denotes the set of resources that are affected by the PMI.
- U - Utilization set. This set indicates the hardware resources that are used by the PMI. Since each field of the control word in LUCAS directly controls a resource, the set serves the additional purpose of showing which fields are blocked by the MOs of the PMI.

- T - Transformation set. This set indicates the hardware resources that are used by the PMI, but which not necessarily inhibits two conflicting PMIs from being put in the same MI during code improvement.
- F - Forward chain is a pointer to a PMI which must reside in the next sequential MI.
- B - Backward chain is a pointer to a PMI which must reside in the immediately preceeding MI. F and B are used to link pipelined operations to assure that they are executed in consecutive order.
- J - Jump. Indicates if the PMI is a microcode branch instruction.

As an example, consider the following statement:

ADMA(S,DIRECT)

The meaning of ADMA (Add Multiplexer All) is that the data bit coming from the multiplexer should be added to the R register in all PEs (see Figure 4.2). The C register is used for both incoming and resulting carry. The variable S defines the bit address in the PE memory where the operand is located and DIRECT specifies that no permutation of data is done. The semantic actions which translate the statement are shown in Figure 5.9 (a) and the resulting PMIs are described in Figure 5.9 (b).

	PMI_k	$L = \text{undefined}$ $M = \{TBZ, RB_i\}$ $I = \{RB_i\}$ $O = \phi$ $U = \phi$ $T = \{\text{Adr-Proc}\}$ $F = \text{pointer to } PMI_{k+1}$ $B = \text{nil}$ $J = \text{false}$
	PMI_{k+1}	$L = \text{undefined}$ $M = \{\text{ADMA}, \text{DIRECT}\}$ $I = \{R, C\}$ $O = \{R, C\}$ $U = \{\text{PE-ALU}\}$ $T = \phi$ $F = \text{nil}$ $B = \text{pointer to } PMI_k$ $J = \text{false}$
Gen(TBZ)		
Datajoin(RB,i)		
Chain		
Gen(ADMA)		
Join(DIRECT)		
(a)		(b)

Figure 5.9 Example of PMIs.

5.5 CODE IMPROVEMENT

The code produced so far is purely sequential with one single activity per PMI. As mentioned above, it can be processed by the final assembler phase of the compiler, but the resulting microcode would be very inefficient. The code improvement layer deals with the reorganization of the intermediate code in order to obtain a higher degree of concurrency between microoperations.

The code produced by the code improvement layer is a structure

similar to the PMI list, where the elements are MIs instead of PMIs. An MI is represented by the seven-tuple: $MI = \langle L, P, I, O, U, T, J \rangle$ where P denotes the set of PMIs that are contained in the MI and where L, I, O, U, T and J have the same meaning as in the PMI tuple.

Two methods for reorganizing the code are used. The first of these preserves the order of activities. If operation j preceeds operation k in the PMI list, either j still preceeds k in the resulting code or they are located in the same MI. We refer to this method as packing of the microcode. Packing is useful for debugging microprograms. The order of execution follows the order of operations in the source code and still a fairly efficient code is used as compared to executing the PMI list. A more efficient microcode is obtained if the reorganization of the intermediate code allows a change of the relative order between the PMIs. Such a scheme which guarantees that the resulting MI list is semantically equivalent to the original PMI list is called a code compaction.

5.5.1 Packing

The packed MI list is initially empty. Starting with the first PMI, subsequent PMIs are merged into one single microinstruction which is appended to the MI list according to the algorithm in Figure 5.10. Packing can be performed during the first compilation pass, in which case no list of PMIs has to be created.

```

generate empty MI;
get next PMI from list;
EMIT:=false;
CONFLICT:=false;
while PMI list not empty do
begin
  if LPMI is defined then CONFLICT:=true
  (* PMI has label *)
  else
    if  $U_{MI} \cap U_{PMI} \neq \emptyset$  then CONFLICT:=true
    (* Resource conflict *)
    else
      if  $O_{MI} \cap I_{PMI} \neq \emptyset$  then CONFLICT:=true
      (* Data dependency: MI must execute before PMI *)
      else
        if  $T_{MI} \cap T_{PMI} \neq \emptyset$  then
          begin
            consult transformation table;
            if transformation possible then
              transform and add PMI to MI
            else CONFLICT:=true;
          end else add PMI to MI;
          if JPMI then EMIT:=true;
          if FPMI  $\neq$  nil then EMIT:=true;
          if EMIT or CONFLICT then
            begin
              append MI to list;
              generate empty MI;
              if CONFLICT then add PMI to MI;
              EMIT:=false; CONFLICT:=false;
            end;
          get next PMI;
        end {while}
      if MI not empty the append MI to list;

```

Figure 5.10 Packing algorithm

When a new MI is generated its tuple elements are empty sets. Adding new PMIs to the MI consists of forming the union between the sets of

the MI and the PMI and to assign the resulting sets to the MI tuple. The meaning of the two Boolean variables EMIT and CONFLICT is as follows: EMIT is set true when the last PMI was added to the MI but when a new MI must be generated for the next PMI. CONFLICT is set true when the last PMI could not be added to the MI.

In some cases it is possible to transform two consecutive activities in one resource into one single operation as follows:

$$\begin{array}{l} \text{PMI}_i \\ \text{PMI}_{i+1} \end{array} \quad \text{--->} \quad \text{Transformed PMI}_i$$

An attempt to perform such a transformation is made when the following condition is valid between the current PMI and the last generated MI:

$$(T_{\text{PMI}} \cap T_{\text{MI}} \neq \emptyset) \wedge (U_{\text{PMI}} \cap U_{\text{MI}} \neq \emptyset)$$

To see if the transformation is possible, a transformation table is consulted. In LUCAS, transformations are defined for certain operations on the Address Processor. Table 5.1 shows the transformation table for the Address Processor. The left column is the sequence $\text{PMI}_i, \text{PMI}_{i+1}$. The next column gives the transformed PMI_i under the condition in the rightmost column. The first entry in the table shows that the sequence TBZ, LDBD can always be transformed to ALDBD. The second entry shows that the sequence TBZ, LDBA can be transformed to LDBA under the condition $I_i = I_{i+1}$. This means that for example

$$\begin{array}{l} \text{TBZ RB4} \\ \text{LDBA RB2 RA4} \end{array} \quad \text{--->} \quad \text{LDBA RB2 RA4}$$

is a valid transformation, since $I_i = \{R4\}$ and $I_j = \{R4\}$ (the register R4 in the Address Processor is the input operand of both operations). However

TBZ RB4

LDBA RB4 RA2

cannot be transformed since $I_i \neq I_j$. The sequence TBZ,INCB can always be transformed into AINCB, for example

TBZ RB4

INCB RB6

---> AINCB RB6 RA4

is a valid transformation. And so on.

Initial PMI sequence	Resulting PMI	Condition
TBZ LDBD	ALDBD	always
TBZ LDBA	LDBA	$I_i = I_j$
TBZ INCB	AINCB	always
TBZ DECB	ADECB	always
TBZ ADAD	AADAD	$I_i = I_j$
TBZ ADBA	AADBA	$I_i = I_j$
TBZ SUBBA	ASUBBA	$I_i = I_j$
LDBD TBZ	LDBD	$O_i = I_j$
LDBD TBZ	ALDBD	$O_i \neq I_j$
LDBA TBZ	LDBA	$(I_i = I_j) \vee (O_i = I_j)$
INCB TBZ	INCB	$O_i = I_j$
INCB TBZ	AINCB	$O_i \neq I_j$
DECB TBZ	DECB	$O_i = I_j$
DECB TBZ	ADECB	$O_i \neq I_j$
ADAD TBZ	ADAD	$O_i = I_j$
ADBA TBZ	ADBA	$O_i = I_j$
ADBA TBZ	AADBA	$(I_i = I_j) \wedge (I_i \neq O_i)$
SUBBA TBZ	SUBBA	$O_i = I_j$
SUBBA TBZ	ASUBBA	$(I_i = I_j) \wedge (I_i \neq O_i)$

Table 5.1 Transformation for Address Processor operations

5.5.2 Compaction

The code compaction problem has been studied during the last decade and several algorithms have been proposed. Initially, only local compaction was considered, i. e. the compaction of basic blocks. A basic block is defined as a sequence of consecutive operations (PMIs in our terminology) which may be entered only at the beginning and which is jump-free, except possibly at its end.

It has been shown [Lan-1] that the problem of finding the optimal solution to the local compaction problem is NP complete. However, several non-optimal algorithms with less computational complexity have proved to be very useful in practice. Many of these algorithms were first proposed for rather simplified models of the microprogrammed architecture. Current research is directed to extending these models and also to performing compaction over larger parts than basic blocks. Examples of research areas are:

- * Polyphase systems, where microoperations occupy resources during a part of the machine cycle.
- * Multicycle operations, where one microoperation occupies resources during several consecutive machine cycles. [Tok-1, Mez-1].
- * Delayed resource binding. When alternative resources may be used for an operation, it is often advantageous to delay the decision of which one to use until the compaction phase. [DeW-1, Lan-1].
- * Global compaction. Compaction is not limited to basic blocks. [Fis-1, Tok-1].
- * Transformability. The meaning of this was explained in the previous section. To our knowledge no investigation has reported on this topic.

In the compiler we deal with monophase machine cycles and with multicycle operations (we call them pipelined operations). Resource binding is performed by the parser and not by the compaction phase. The code compaction is restricted to basic blocks. In the following we will assume that the PMI list has been separated into basic blocks and that the list in consideration constitutes one basic block.

An important concept in the compaction process is the data dependency relation between the PMIs. Let i and j be two PMIs where i precedes j in the original PMI list. We say that j is data dependent on i , written $\langle i \text{ dp } j \rangle$, if there is a data interaction between i and j , i. e. if any of the following conditions hold:

$$\begin{aligned} O_i \cap I_j &\neq \emptyset \\ I_i \cap O_j &\neq \emptyset \\ O_i \cap O_j &\neq \emptyset \end{aligned}$$

It is clear that the compaction algorithm must assure that the data dependency relations are kept intact when the MI list is produced.

If $\langle i \text{ dp } j \rangle$ and there is no sequence of $p_1, p_2, \dots, p_k$ such that

$$\langle i \text{ dp } p_1 \rangle \wedge \langle p_1 \text{ dp } p_2 \rangle \wedge \dots \wedge \langle p_k \text{ dp } j \rangle$$

then we say that j is directly data dependent on i , written $\langle i \text{ ddp } j \rangle$. From this definition it is possible to define a data dependency graph (DDG) which represents a partial order over the set of PMIs contained in a basic block. In the DDG, a node represents a PMI, and an edge from node i to node j exists if $\langle i \text{ ddp } j \rangle$. Figure 5.11 shows the DDG for the basic block $\{1, 2, 3, 4, 5, 6, 7\}$ with the following partial order over the elements:

$$\begin{aligned} &\langle 1 \text{ ddp } 3 \rangle, \langle 3 \text{ ddp } 4 \rangle, \langle 3 \text{ ddp } 6 \rangle, \\ &\langle 6 \text{ ddp } 7 \rangle, \langle 2 \text{ ddp } 5 \rangle, \langle 5 \text{ ddp } 7 \rangle \end{aligned}$$

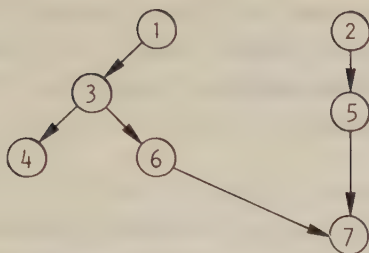


Figure 5.11 Data dependency graph

One of the first algorithms presented for local compaction is the Critical Path algorithm, which was introduced in 1974 [Ram-1]. The idea of the algorithm is to find the longest path in the DDG. The PMIs which are on this "critical" path must be assigned to consecutive MIs in order for the MI list to be optimal. The first step of the algorithm consists in locating the critical path, produce an MI list of appropriate length and assigning the PMIs on the critical path to the MIs in the list. For the DDG of Figure 5.11 the critical path consists of the nodes 1,3,6 and 7.

Next, the remaining PMIs are considered. Each of these may be assigned to MIs within a certain time frame. In Figure 5.11 the element 4 must be assigned to a time frame later than element 3. i. e. it could be assigned to the same MI as either element 6 or 7. Due to resource conflicts it may be necessary to split certain MIs on the critical path during this process.

Examples of implementations using the critical path algorithm are the Strum system [Pat-1] and MORIF [Tok-1].

A Branch and Bound algorithm has been described in [Yau-1]. In the algorithm a tree, where the nodes are possible MIs, is built from its

root. To create a new node in the tree, the PMIs whose predecessors in the DDG have been assigned to nodes in the MI tree are considered. A new branch is created whenever more than one node may be formed by combining these PMIs.

The complete tree represents every possible ordering of the PMIs and the shortest path from the root to a leaf corresponds to an optimal ordering. It has been shown that this algorithm, which leads to an optimal solution of the compaction problem, is NP complete [Lan-1]. Different heuristics have been proposed which reduce the computational complexity by pruning the tree, but which lead to non-optimal solutions.

The First-Come First-Served (FCFS) algorithm which was described by Dasgupta and Tartar [Das-1] is different from the other algorithms in that the PMIs are considered in source code order and that the DDG does not have to be calculated in advance. The algorithm is as follows:

- 1) The PMIs are added to an initially empty list of MIs. Every PMI is moved up as far as it can go in the list using the rule that a PMI can be moved ahead of an MI if it is not data dependent on any of the PMIs in that MI. When a data dependency occurs, the PMI has reached its rise limit.
- 2) Search downwards in the list to find an MI where the PMI may be added with respect to resource conflicts. If no such MI exists, a new MI containing the PMI is appended to the list.
- 3) If no rise limit was found in 1), the PMI was not data dependent on any PMI in the MI list and it may be added to any list element. If there is no MI to which the PMI can be added without a resource conflict, the PMI is placed in a new MI at the top of the list. Placing it at the top rather than at the bottom of the list will keep it from blocking any subsequent PMI due to a data dependency restriction.

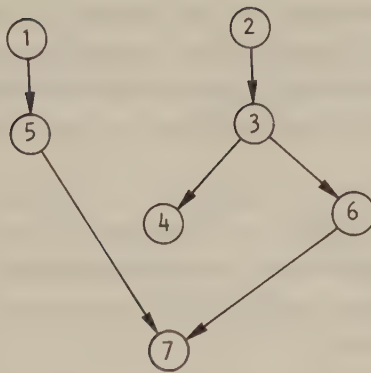
Practical experiments [Dav-1] have shown that the code obtained is of

the same quality as the code produced by the other non-optimal algorithms. In addition, the FCFS algorithm has advantages in both speed and simplicity. Implementations of the algorithm are described in [Mez-1, Bab-1]. The code compaction in our compiler is based on this algorithm.

The following comment can be made to the algorithm: The same argument which is used when placing a PMI at the top of the list in 3) can be used even if a data dependency occurs. This means that 3) can be omitted if 2) is changed to:

- 2) Search downwards in the list to find an MI where the PMI may be added with respect to resource conflicts. If no such MI exists, a new MI containing the PMI is inserted at the rise limit, or at the top of the list if no data dependency occurred in 1).

Figure 5.12 shows an example of how the algorithm works. (a) shows the DDG, (b) states the resource conflicts between the PMIs (1 c 2 means that a conflict exist between nodes 1 and 2) and (c) shows how the MI list is built while adding the PMIs in source order (1,2,3,4,5,6,7). In the example, the resulting microcode is optimal. Four MIs is the minimum number since this is the length of the longest path in the DDG.



(a)

1 c 2
5 c 3

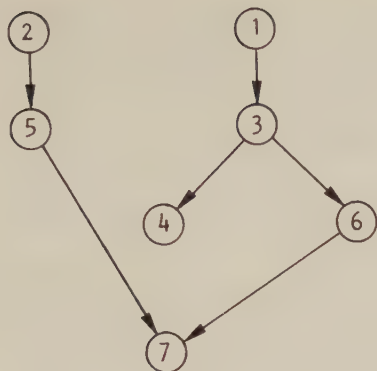
(b)

MI		PMI added						
		1	2	3	4	5	6	7
1		1	2	2	2	2	2	2
2			1	1,3	1,3	1,3	1,3	1,3
3					4	4,5	4,5,6	4,5,6
4								7

(c)

Figure 5.12 Execution of the FCFS algorithm. (a) Data dependency graph. (b) PMIs with resource conflicts. (c) Building the MI list

The FCFS algorithm does not always produce an optimal solution. Figure 5.13 shows how renumbering the nodes in the DDG (i. e. changing the source code order of the PMIs without changing the data dependency relations) influences the result. In Figure 5.14 an additional resource conflict between PMIs 4 and 6 in the renumbered graph of Figure 5.13 results in an MI list of length six.



(a)

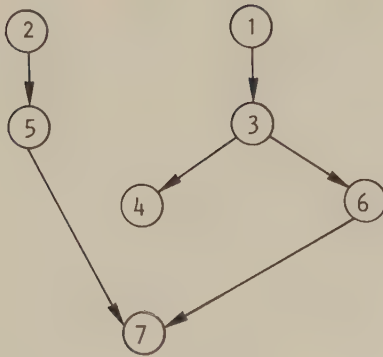
1 c 2
5 c 3

(b)

MI	I	PMI added						
		1	2	3	4	5	6	7
1	I	1	2	2	2	2	2	2
2	I		1	1	1	1,5	1,5	1,5
3	I			3	3	3	3	3
4	I				4	4	4,6	4,6
5	I							7

(c)

Figure 5.13 Execution of the FCFS algorithm after renumbering the nodes of the DDG.



(a)

1	c	2
5	c	3
4	c	6

(b)

MI	PMI added							
	1	2	3	4	5	6	7	
1	1	2	2	2	2	2	2	
2		1	1	1	1,5	1,5	1,5	
3			3	3	3	3	3	
4				4	4	4	4	
5						6	6	
6							7	

(c)

Figure 5.14 Execution of the FCFS algorithm after renumbering the nodes of the DDG and adding a resource conflict between nodes 4 and 6.

We will now introduce an extension to the FCFS algorithm which is used in our compiler. Using this Extended FCFS (EFCFS) algorithm, the microcode obtained is often more compact than with the original FCFS and never less compact. The EFCFS scheme is as follows:

- 1) Backward compaction. Create an MI list from the PMI list using the FCFS algorithm.

- 2) Forward compaction. Starting at the end of the MI list and working backwards, one PMI is selected at a time. This PMI is now moved down as far as it can go in the list using the rule that it may be moved below an MI if no PMI in this MI is data dependent on the PMI in consideration. When this happens, the PMI has reached its lower limit.

Now the list is searched backwards to find an MI where the PMI can be added with respect to resource conflicts. If no such MI exists, the PMI is left in its original position and the next PMI is considered.

- 3) If during 2) all the PMIs of an MI are removed, the MI is removed from the list.

Figure 5.15 (a) and (b) show the result of applying the EFCFS algorithm to the examples of Figure 5.13 and Figure 5.14 respectively. In the first case the number of MIs is reduced from five to the optimal four and in the second case from six to four, which is also optimal in this case.

Practical tests of the compiler have shown that the extensions to the FCFS algorithm significantly reduce the number of microinstructions in the final microcode.

MI	I	PMI considered for forward compaction						
		initial						
		MI list	6	4	3	5	1	2
1	I	2	2	2	2	2	2	1
2	I	1,5	1,5	1,5	1,5	1	1	2,3
3	I	3	3	3	3	3	3	5,6
4	I	4,6	4,6	6	6	5,6	5,6	4,7
5	I	7	7	4,7	4,7	4,7	4,7	

(a)

MI	I	PMI considered for forward compaction						
		initial						
		MI list	6	4	3	5	1	2
1	I	2	2	2	2	2	2	1
2	I	1,5	1,5	1,5	1,5	1	1	2,3
3	I	3	3	3	3	3	3	5,6
4	I	4	4	6	6	5,6	5,6	4,7
5	I	6	6	4,7	4,7	4,7	4,7	
6	I	7	7					

(b)

Figure 5.15 Application of the EFCFS algorithm to the examples of Figure 5.13 and Figure 5.14.

In some cases it is possible to add the PMI to the MI immediately above the rise limit or to the MI immediately below the lower limit. This happens when the data dependency relations between the PMI and the MI are weak dependencies.

Let i and j be two PMIs, where j is directly data dependent on i

$\langle i \text{ ddp } j \rangle$. If for every data interaction between them, PMI_i finishes with the interacting resource before PMI_j starts to use it then they are weakly dependent, written $\langle i \text{ wdp } j \rangle$.

In LUCAS every change of state takes place at the end of the machine cycle. This means that if $\langle i \text{ ddp } j \rangle$ and the only interaction between i and j is defined by $I_i \cap O_j \neq \emptyset$ then $\langle i \text{ wdp } j \rangle$.

This is illustrated in Figure 5.16, which shows two PMIs with data interaction via the resource R . In (a) $O_i \cap I_j \neq \emptyset$ and $\langle i \text{ ddp } j \rangle$. In (b) $I_i \cap O_j \neq \emptyset$ which means that $\langle i \text{ wdp } j \rangle$. In (c) $O_i \cap O_j \neq \emptyset$ and $\langle i \text{ ddp } j \rangle$.

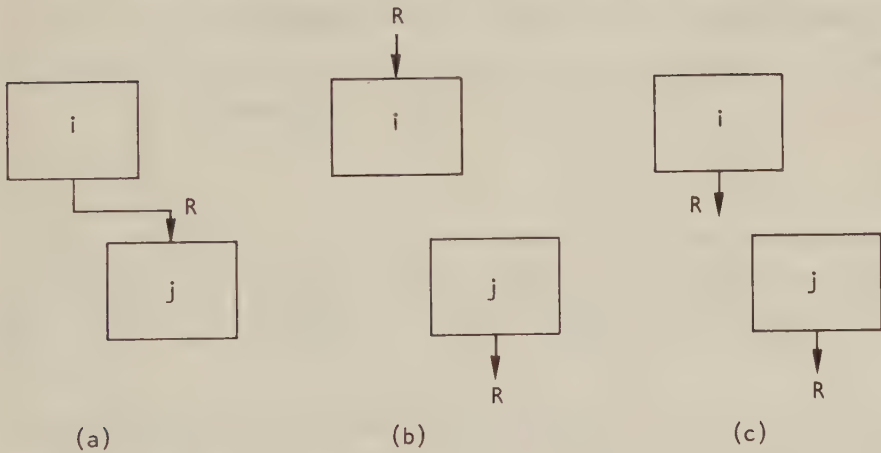


Figure 5.16 Data dependency relations in LUCAS. (a) $\langle i \text{ ddp } j \rangle$, (b) $\langle i \text{ wdp } j \rangle$, (c) $\langle i \text{ ddp } j \rangle$.

5.5.3 Transformation and Pipelining

So far we have neither discussed pipelined PMIs nor the transformation of PMIs as defined by their transformation set. When the addition of a PMI to an MI is considered and the following condition holds:

$$(U_{MI} \cap U_{PMI} = \emptyset) \wedge (T_{MI} \cap T_{PMI} \neq \emptyset)$$

the transformation table is consulted to decide if a transformation is possible. In some cases it is important that the transformation table is consulted for the correct sequence of PMIs. The correct sequence in different situations is given in Figure 5.17. Normally, the two transformed PMIs are replaced by the result of the transformation. However, if one of the PMIs is part of a pipelined operation (as discussed below), both this PMI and the transformed PMI are kept in the MI.

Transformation situation	Sequential order of PMIs
<u>Backward Packing:</u>	
Transformation with MI above rise limit (weak dependence)	$PMI_{in\ MI}, PMI_{added}$
Other transformation	Any order. Try both.
<u>Forward Packing:</u>	
Transformation with MI below lower limit (weak dependence)	$PMI_{added}, PMI_{in\ MI}$
Other transformation	Any order. Try both.

Figure 5.17 Sequential order to use when consulting the transformation table.

Pipelined PMIs must be taken care of both during backward compaction and during forward compaction. The methods used in these two cases are different but similar and only backward compaction will be described.

The fact that a PMI is part of a pipelined operation is discovered when the Backward chain, B , of the last added PMI is defined and points to another PMI. Let Y be the last and X the next but last added PMI. Further, let j be the MI where Y was added and i the MI where X was added. (Note that X and i are defined by the Backward chain of Y , B_Y .) Pipelining is limited to two levels in LUCAS, which means that B_X is undefined. Figure 5.18 illustrates the situation. Due to data dependencies it is clear that $i < j$.

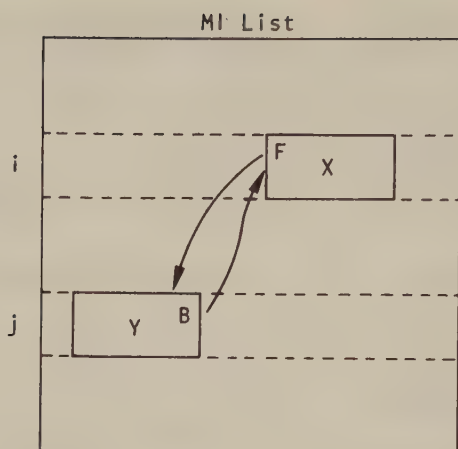


Figure 5.18 Two pipelined PMIs located in MIs i and j of the MI list.

```

if j=i+1 then exit;
release X from MIi;
attempt to add X to MIj-1;
if possible then exit;
release Y from MIj;
j:= next MI where Y can be added;
while such j exists do
begin
  attempt to add X to MIj-1;
  if possible then
  begin
    add Y to MIj;
    exit;
  end;
  j:=next MI where Y can be added;
end;
attempt to add X to the last MI of the list;
if not possible then append a new MI containing X to the list;
append a new MI containing Y to the list;

```

Figure 5.19 Taking care of pipelined PMIs during Backward compaction.

In Figure 5.19 an algorithm is given which assures that two pipelined PMIs keep their consecutive order in the MI list. When it is necessary to release a PMI from its location in the MI list, retransformation of transformed PMIs may be needed. To make this possible, a PMI which is part of a pipelined operation is always kept in its original form in the MI. Upon exit from the algorithm any such PMI should be removed.

5.5.4 Other Improvements

Standard code improvement techniques from the theory of language translation could in principle be applied in the compiler. The most important such techniques are aimed at removing code from loops to increase the execution speed. However, the importance of using these techniques depends on the presence of removable code in the loops. This code originates from either the programmer or the compiler. In the first case, it can be avoided by careful programming. In the second case, the code is very often produced when translating expressions and array references. Due to the simplicity of our source language, where expressions are semantically close to the microcoded architecture and where no arrays are included, compiler generated removable code is rare.

For the reasons mentioned above, the code improvement phase does not include any "standard" optimizations. However, two heuristic methods are used prior to the local code compaction. The first of these deals with "jumps-over-jumps" and "jumps-to-jumps". Especially "jumps-over-jumps" are frequently generated by the compiler. Consider the translation of the following statement inside a loop construct:

```
if p=0 then exit
```

Using the rules of Figure 5.4 and Figure 5.6, we arrive at the sequence of semantic actions shown in Figure 5.20 (a). The resulting intermediate code is listed in Figure 5.20 (b). We see that the translation of the if-statement results in a jump over the exit-statement which is also translated into a jump. In Figure 5.20 (c) the "jumps-over-jumps" rule has been applied and one of the jumps has been removed by complementing the test condition of the if-generated branch.

```

Gen(TBZ)
Datajoin(RB,p)
Chain
Gen(ZAP)

Join(ComplementTest)
Join(CJP)
i:=NEXTLABNO
Newlabelout

Gen(CJP)          TBZ RBp
ExitLabel         CJP NZAP Li          TBZ RBp
                  CJP Out_of_loop      CJP ZAP Out_of_loop

Oldlabelin(i) Li: ...

(a)                (b)                (c)

```

Figure 5.20 Jumps-over-jumps elimination

The second heuristic method for code improvement is based on the following observations:

- * In tight loops, as in bit-serial arithmetic processing, it is extremely important that the Address Processor is fully utilized. Practical tests show that resource conflicts between Address Processor operations are very common during code compaction.
- * The Address Processor is often used in pipelined operations where the first MI produces a bit address to the PE memories and the following MI uses this address in a PE instruction. Very often the first MI of a loop contains an Address Processor instruction, e. g. TBZ, which is used in the second MI to read out a bit from one of the operands. On the other hand, the last MI in

the loop contains often no Address Processor instruction since no MI follows.

The method we use is aimed at reducing the size of the loops. This is done by copying the Address Processor instruction from the first PMI in the loop to the last one and thereby advance the target of the backwards branch by one PMI. The algorithm used for loop size reduction is as follows:

- 1) Locate a loop in the PMI list (=a backwards branch). It may consist of one or several basic blocks.
- 2) If the first PMI in the loop is a TBZ or a PLADR microoperation to the Address Processor then:
 - i) Join a copy of this PMI to the last PMI of the loop. (This is always possible since the last PMI is a jump instruction, which never affects the Address Processor.)
 - ii) Define a label on the second PMI of the loop.
 - iii) Change the branch target at the end of the loop to this new label.
- 3) Search for the next loop in the PMI list. If found then go to 2).
- 4) Remove unreferenced labels from the PMI list.

Figure 5.21 demonstrates the impact the method has on a typical bit-serial loop. (A "typical" bit-serial computation loop contains the following steps: get one bit from the first operand, get one bit from the second operand, perform a Boolean operation on the bits, store the result bit.) (a) shows the source code of the loop, (b) is the PMI list produced by the compiler, (c) is the resulting microcode after local code compaction of the PMI list. This results in a loop containing four microinstructions.

Figure 5.21 (d) shows the PMI list after loop size reduction has been

applied and (e) is the resulting microcode after code compaction of the PMI list in (d). This loop contains three microinstructions, which is the minimal length.

```

iterate b times
begin
  LRMA(a,direct);
  ADMA(b,direct);
  WRRR(c);
  a:=a+1; b:=b+1; c:=c+1;
end;

```

(a)

<pre> LDCT b-1 LOOP: TBZ RBa LRMA DIRECT TBZ RBb ADMA DIRECT TBZ RBc WRRR INCB RBa INCB RBb INCB RBc RPCT LOOP </pre>	<pre> LDCT b-1 LOOP: AINCB RBa RAa AINCB RBb RAb LRMA DIRECT AINCB RBc RAc ADMA DIRECT RPCT LOOP WRRR </pre>
--	---

(b)

(c)

<pre> LDCT b-1 TBZ RBa LOOP1: LRMA DIRECT TBZ RBb ADMA DIRECT TBZ RBc WRRR INCB RBa INCB RBb INCB RBc RPCT LOOP TBZ RBa </pre>	<pre> LDCT b-1 TBZ RBa LOOP1: AINCB RBa RAb LRMA DIRECT AINCB RBb RAc ADMA DIRECT RPCT LOOP WRRR AINCB RBc RAa </pre>
--	--

(d)

(e)

Figure 5.21 Application of loop size reduction on a tight loop

5.6 CONCLUSIONS

Two of the design objectives for the microprogramming language which were stated in Chapter 4 were that the language should have a machine independent high-level like structure and that it should allow efficient microcode to be produced. In this chapter we have seen how various constructs of the language can be translated and we may now attempt to estimate how well the architecture of LUCAS is adapted to the language. The question is: does the high-level properties of the language influence the efficiency of the generated code in an unfavourable way? Another question is: how good is our translation method and how could it be improved?

The influence of the block structure on the microcode is minimal. One consequence is that the number of variables which must be considered during translation is reduced, which allows a simple method for register allocation. The inclusion of subroutines is straightforward since subroutine linking is directly supported by the Sequencer. The use of local variables in subroutines leads to some overhead, especially in nested subroutine calls. However, a more elaborate scheme for register allocation could most certainly reduce this overhead at the expense of a global data analysis during code generation.

We have seen that the control statements of the language are efficiently translated into conditional branches and instructions for the Testmultiplexer. The iterate construct, which is frequently used in bit-serial loops, results in a very efficient utilisation of the resources. Assignments and conditions in the language are primitive and are semantically close to the capabilities of the Address Processor.

The code compaction algorithm which is used in the compiler gives very good results and optimal code is most often obtained for the

basic blocks. In bit-serial processing most of the time is spent in fairly tight loops which are often represented by basic blocks in the intermediate code. The effects of a global compaction method would probably not be important enough to justify the amount of work needed for its implementation. Further improvements of the microcode could probably be obtained by "loop jamming", where the bodies of several loops are merged into one single loop, as this could result in basic blocks of larger size.

Part 4. High-Level Programming

Chapter 6 LANGUAGES FOR PARALLEL MACHINES - A SURVEY

6.1 INTRODUCTION

This chapter is a survey of programming languages which are suitable for parallel computing on SIMD machines. Common to all these is the possibility to declare and to operate on distributed data, i. e. multi-element data where the single elements are located in different processing elements or in different words of an associative memory. Languages, which are oriented towards parallel process computation, such as Concurrent Pascal, Modula-2 or Ada, are not considered here, since they support a completely different form of parallelism than the SIMD type.

Highly parallel machines of type associative array computers are often programmed in assembly like languages. The reason is that machines like these tend to be unique, and they differ in several important aspects. For example:

* Number of processing elements. This reaches from 64

elements on ILLIAC IV [Bar-1] to over 16,000 on the MPP [Bat-5].

- * Complexity of the processing elements. On ILLIAC IV, the PEs are powerful pipelined processors, which perform floating point arithmetic in hardware. At the other end of the spectrum we find machines like STARAN [Bat-3], the MPP, DAP [Red-1] and LUCAS, which are all bit-serial processors.
- * Interconnection Structure. The topology of the processing array is defined by the PE interconnection network. This topic was discussed in Section 1.6.

It is extremely important that the extra speed offered by a parallel computer is utilized since this is its only "raison d'être". We note that most of the languages, which have been proposed for highly parallel computers, are directed towards a specific machine. This reflects the fact that if a high level language is to be used, it must fit the architecture well to be efficient.

6.2 BASIC CHARACTERISTICS

The first attempt to specify high level language programming of associative processors is found in [Fin-1]. In this early paper a library of FORTRAN subroutines is presented which allows instructions that control a simulated associative processor in FORTRAN programs. This Associative Memory-Parallel Processor Language-I (AMPPL-I) does not provide a high level specification of operations on the data in the associative memory, but can be described as an assembly language for a fictive associative processor, imbedded in FORTRAN programs.

It is interesting to note that the proposed language is not intended to be implemented on a parallel machine. The idea is to give the programmer of a standard sequential computer an alternative view of the machine; a conceptual view that would be natural for certain applications. The author says:

"It is more than likely that its [the language's] implementation will not yield a system fast enough to be capable of providing real time computing support for such tasks as radar tracking or industrial control in a rapidly varying environment. For many applications, however, the language proposed may represent two advantages: (a) it would decrease the length of written programs, and (b) it would simplify the writing and understanding of programs."

Another early paper describing a high level language for associative computing is [Fel-1]. As was the case with AMPPL-I, the language proposed here, LEAP, is not intended to be implemented on a parallel computer. In LEAP the programmer can declare items in an associative universe (=associative memory). A three-tuple of items forms an association and has the form:

<attribute,object,value>

which is syntactically expressed as a.o=v (for example father.john=bill). It is possible to create sets of associations and new sets may be formed by specifying operations on previously defined sets and groups of sets.

The most important parallel operation which can be specified in LEAP programs is the associative search, and the authors argue that this is accomplished more efficiently using hash-coded memories than associative memories. In the limited forms of associations that is assumed, this is probably true.

With the introduction of ILLIAC IV, the need for new high level languages arose. Several languages were proposed of which at least three were implemented and used: CFD [Ste-1] - based on FORTRAN,

Glypnir [Law-2, Law-3] - based on Algol 60 and IVTRAN [Mil-1] - based on FORTRAN. In the following sections the latter two will be presented in more detail since we consider them to be the most important ones. Glypnir because it has been the most widely used language amongst the ILLIAC IV users [Thu-1, Slo-1] and IVTRAN because of its rather unique definition and implementation with the compiler detecting parallelism in programs which are written in sequential code. The first language proposed for ILLIAC IV, however, was none of the three mentioned; it was called TRANQUIL [Kuc-1]. A preliminary specification of TRANQUIL was made at the same time as the design of the ILLIAC IV system.

The language is based on Algol 60, with some minor deletions, and includes extensions for parallel execution of statements and declaration of arrays that are stored over the PEs. Using its data declarations, it is possible to specify the layout of variables in the PE memories (STRAIGHT or SKEWED mapping). A PARTITION declaration is used to partition previously declared arrays in several ways and to form subarrays. Parallel execution of one (e. g. a for-loop) or several statements is specified by means of a SIM statement:

```
SIM BEGIN (S1; S2; ... Sn) END
```

The implementation of the language was never completed.

Special purpose languages have also been described for use in important application areas for associative array processors - such as image processing and database management. Pixal [Lev-1] consists of parallel extensions to Algol 60 and is directed towards low level picture processing. Its FRAME construct is used to specify an environment to each cell in the array upon which operations are performed.

Another language which is suitable for image processing is PascalPL [Uhr-1]. It is defined in terms of extensions to Pascal. Parallel operations (instructions which are executed simultaneously by two or more processing elements) can be included, but only within procedures which are declared as parallel. A STRUCTURE specification, similar

to the FRAME construct in Pixa1 is defined. The use of the structure concept is illustrated in the following example:

```
!!set newimage := image( + (0:0*4,0:1*-1,1:0*-1,0:-1*-1,-1:0*-1));
```

This assignment statement specifies that all the elements in the previously declared array NEWIMAGE are assigned new values in parallel. The new values are given by the "compound of array-specifications" which follows the assignment operator (in the example this "compound" is reduced to one array-specification). An "array-specification" consists of an array name followed by a structure. The structure in the example gives a weighted sum of the neighbour elements in the array: each element in the new array is assigned the value of the corresponding element in the original array multiplied by four, minus the values of the four horizontal and vertical neighbouring elements.

Syntactically, the language suffers from the fact that too much attention has been given to allow a simple translation to Pascal, by the use of a preprocessor. So far, PascalPL has not been implemented on a parallel machine.

An important part of any language for parallel computing is the indexing scheme, i. e. the mechanism for selecting the set of operands to be operated on in parallel. In most languages, the distributed data is declared in the form of arrays whose elements are allocated to different processing elements. It is likely that the programmer needs to specify operations on subsets of the parallel data arrays. In a conventional language this can be expressed:

```
A[5] := A[5]*2
```

or

```
for i:=5 to 12 do
  A[i] := A[i*2];
```

It is also necessary to include alignment constructs in the language. This is used in statements which involve communication of data between different processing elements, such as:

```
for i:=5 to 12 do  
  A[i]:=A[i+2];
```

A language which implements a very flexible indexing scheme is APLISP (A Parallel Language for Image and Speech Processing) [Mue-1]. Here the parallel arrays are treated as sets and subsets are chosen by "index sets". Operations on the index sets, such as the Cartesian product, intersection or concatenation, allow a powerful indexing of the operands, including both alignment and "window specification" similar to the ones defined in Pixa1 and PascalPL. Representing arrays by sets has the advantage that highly parallel operations can be specified without any preferred order of evaluation, when no such order is necessary. This clearly allows a great degree of freedom for the compiler to exploit parallelism.

As part of the Phoenix Project [Fei-1], a language called Actus has been defined [Per-1]. The language, which is based on Pascal, is in many aspects similar to both Parallel Pascal, which will be described in Section 6.3, and to Pascal/L, which is the subject of Chapter 7. However it includes constructs, such as independent indexing in the processing elements, which could not be efficiently implemented on LUCAS, neither the MPP, which is the target computer for Parallel Pascal. Languages for the commercially available machines DAP and STARAN are described in [Fla-1] and [Lan-2]. These languages do not include any concepts other than what have been discussed.

Several high level languages for database management on associative computers have been proposed [Res-1, Bra-1, Lov-1]. Generally these languages are more limited than the ones mentioned above and the ones that will be presented in the subsequent sections. They are mostly used in applications of data base retrieval, where searching is a major operation, whereas no mechanisms for data alignment are needed.

6.3 THREE CASE STUDIES

6.3.1 GLYPNIR

Glypnir [Law-2, Law-3, Lay-1] is based on Algol 60 and is the first implemented language for ILLIAC IV. In the light of the problems encountered with the implementation of the earlier language, TRANQUIL, it was decided that the work on a less ambitious language should be initiated and run in parallel with the TRANQUIL project.

Neither TRANQUIL nor Glypnir is able to detect parallelism where sequential code is specified, but what makes TRANQUIL so much more difficult to implement is that it supports a more general form of parallelism where for instance arrays can be of any size and the compiler is responsible for "squeezing" them to fit the ILLIAC IV memory.

In contrast to this, all distributed data variables in Glypnir have 64 elements in its parallel dimension (ILLIAC IV has 64 processing elements), and it is the programmer's task to map smaller or larger arrays into this form.

Two kinds of variables may be declared: scalars (or CU VARIABLES) and distributed data (PE VARIABLES). Examples of declarations are:

```
CU INTEGER A;  
CU REAL VECTOR B[100];  
PE INTEGER C;  
PE REAL VECTOR D[100];
```

A and B are non-distributed variables which are accessed element by element by the Control Unit. C and D are distributed variables; C with one element in each of the 64 PEs and D with 100 indexable elements

in each PE. When the accessed elements of a distributed variable are located at the same address as seen by the PEs, it is called a sword (superword). A slice is 64 data elements, each located in a separate PE, but not necessarily at the same address. Thus C and DCAJ are swords, while DCCJ is a slice. To access a slice, the index variable is loaded into the index registers which are part of the ILLIAC IV PEs.

Glypnir is block-structured, with the same rules governing the scope of variables as in Algol 60. Pointers are available and a pointer may be declared either in the form of a CU variable or as a PE variable. The latter allows a separate data structure to be built in each PE.

Boolean variables and expressions represent 64 separate TRUE/FALSE values. If X and Y both are PE variables, the Boolean expression X<Y yields 64 values, one for each PE. A Boolean variable or expression is said to be TRUE when all elements are TRUE and FALSE when all elements are FALSE, otherwise it is "mixed".

Control statements are extended, as compared to Algol, in that they may also be used to control parallel execution. Let "BE" stand for Boolean expression and "S" for statement. Then

IF <BE> THEN <S1> ELSE <S2>

results in S1 being executed in PEs where the corresponding elements of BE are TRUE and S2 in the PEs where the elements are FALSE. An equivalent form of IF <BE> THEN <S> is:

FOR ALL <BE> DO <S>

To control iteration in the program flow, Glypnir contains several constructs:

```

LOOP I <- I1,I2,I3 DO <S>
THRU I DO <S>
FOR I <- I1 STEP I2 UNTIL I3 DO <S>
DO <S> UNTIL <BE>
WHILE <BE> DO <S>

```

Two of these will be described.

- * In the statement `FOR I <- I1 STEP I2 UNTIL I3 DO <S>`, the variables `I`, `I1`, `I2` and `I3` can be either scalars or swords. In the case where they all are swords, the statement `S` may be executed a different number of times in each PE, with different start values and with different increments of the control variable `I`.
- * The `WHILE <BE> DO <S>` statement causes the statement `S` to be repeated until `BE` becomes `FALSE`, i. e. takes the value `FALSE` in all 64 elements. The execution of `S` is limited to those PEs where the corresponding element of `BE` is `TRUE`.

The Glypnir compiler, which runs on a Burroughs B6700, is said to generate a relatively efficient code, even though no optimization is included. A factor 1.5 to 3 in execution speed is reported as compared to assembly programming.

6.3.2 IVtran

The ILLIAC IV IVTRAN system [Mil-1, Mil-2] is based on the following assumptions:

- * The presumptive user is accustomed to programming in FORTRAN. The new language must be of FORTRAN type.
- * It should be possible to use the system for existing programs, written in standard FORTRAN.

This resulted in a new language, defined in terms of parallel extensions to FORTRAN. In order to allow standard FORTRAN programs to be used, one phase was added to the compiler, where parts of the source program, which could have been expressed in IVTRAN, are rewritten. This part of the compiler is called the Paralyzer (Parallelism Analyser and Synthesizer) and it produces an IVTRAN program from the FORTRAN source program. The form of the IVTRAN program is either source code (intended for the interested user) or - since the Paralyzer works on an intermediate form of the program which is produced by the parser - in a form suitable for the next compiler phase, the optimizer. Figure 6.1 represents the IVTRAN compiler.

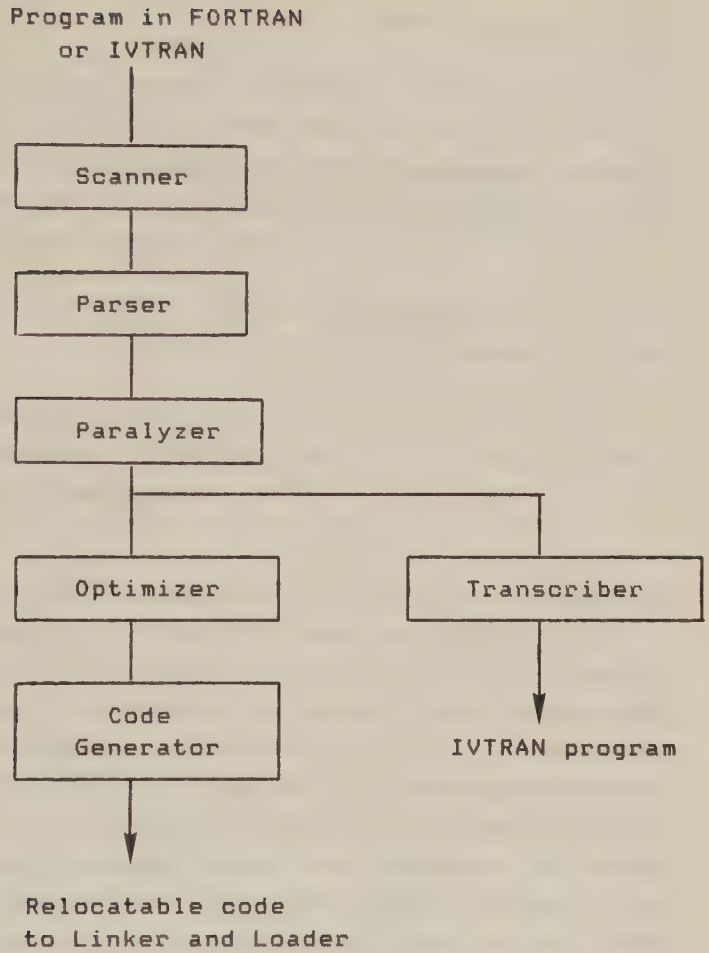


Figure 6.1 The IVTRAN compiler

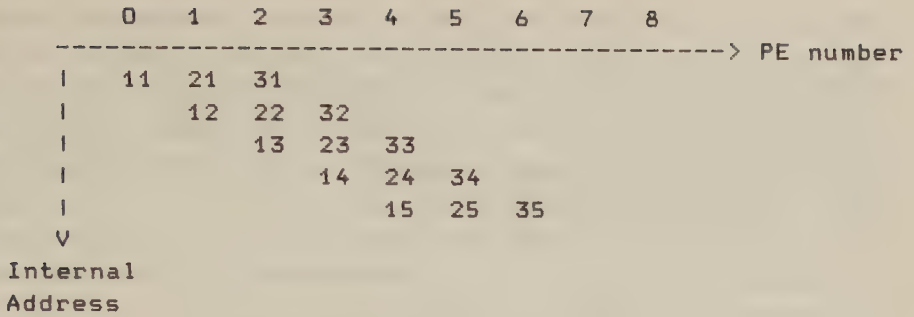
6.3.2.1 Language Elements

The language includes essentially three new constructs which give the user access to parallel processing in the PEs. Two of these, the allocation declarations and the OVERLAP statement, are used to specify the layout of data in the array of PEs, while the third, the DO FOR ALL statement, controls parallel operation of the PEs.

The allocation declaration, which is a part of a DIMENSION or a COMMON statement, has the following syntax:

```
allocation ::= [multi-index {,multi-index}]
multi-index ::= (index{,index}) | #(index{,index}) |
               $(index{,index})
```

Each index in the allocation declaration denotes a subscript position in the declared array. When an index is preferred (denoted by a preceeding \$), an increment by one of this index locates the corresponding array element in the next PE (the PE number is incremented by one) but on the same internal address within the PE as the previous array element. When the index is aligned (denoted by #), an increment by one in the index value does not change the PE number but increments the internal address within the PEs. The default storage scheme is skewed, which means that both the PE number and the internal PE address is incremented when the index value is incremented. Figure 6.2 gives examples of different storing schemes for a 3 x 5 array.



[\$(1),(2)]

11	12	13	14	15	
	21	22	23	24	25
		31	32	33	34 35

[\$(2),(1)]

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35

[(2),#(1)]

11	12	13	14	15	21	22	23	24	25	31	32	33	34	35
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

[(1,2)]

Figure 6.2 Different allocations of an 3 x 5 array

The meaning of the OVERLAP statement is to allow different arrays to occupy the same memory area in order to save space in the PE memories. This is useful if two arrays, A and B, only appears in different parts of the program, in which case A and B can overlap.

The most obvious use for this feature is when the allocation scheme for an array has to be changed to allow efficient processing by the PEs. This can be accomplished by declaring a second array with the wanted allocation and letting the two arrays overlap. If the new array is assigned the value of the original array, a transformation to the new allocation format is performed.

To control parallel operation, IVTRAN includes a DO FOR ALL construct with the following syntax:

```
DO j FOR ALL (i1,...,in)/s
```

where j is the label of the last statement in the loop. i₁,...,i_n are control indices. These are names of variables which will be assigned values according to s during the execution of the loop. s is an n-dimensional logical array expression.

```
DO 10 FOR ALL (I)/[1..3]
10 A(I,2) = A(I,2)*5
```

means that the array elements A(1,2), A(2,2) and A(3,2) are assigned new values simultaneously. When the array A(I,J) appears in a DO FOR ALL (I) - loop, it must have been declared with an allocation of type [\$(1),(2)] to allow fastest possible access to the I-elements, whereas if it appears in a DO FOR ALL (J) - loop, the allocation should be [(1),\$(2)] or [\$(1),\$(2)] (see Figure 6.2).

```
DO 10 FOR ALL (I,J,K)/[1..3].C.[1..7].C.[1..10]
IF (A(I,J,K).LT.0.0) A(I,J,K) = -A(I,J,K)
10 A(I,J,K) = SQRT(A(I,J,K))
```

where .C. is the Cartesian cross-product operator, assigns the absolute value to all elements of A and then replaces this value by its square root. In this case the array A should be declared with the allocation [\$(1,2,3)] to allow simultaneous access to all its elements.

6.3.2.2 The Paralyzer

The parallelism analyser/synthesizer of the IVTRAN compiler [Pre-1] detects parts of FORTRAN programs which can be executed in parallel and rewrites them using the extended constructs of IVTRAN. Parts of FORTRAN programs suitable for transcription into IVTRAN are DO loops containing array references with subscripts depending on the index variables of the DO loop.

The Paralyzer performs its task in three steps: first it locates a DO loop and rewrites it in a form suitable for analysis (the DO loop may be nested to any depth). Next it analyses the parallelism of the loop. The principal part of the analysis is to determine the time-relative ordering of different references to the same array element within the loop. Finally it rewrites the DO loop in the form of a DO FOR ALL statement. Each of the steps will now be described.

The first step, the rewriting of the DO loop before analysis, is governed by the requirements of the analysing part of the Paralyzer. Not all loops can be put in a form suitable for analysis. If a nest of loops cannot be rewritten, an attempt is made to work on the interior nests. Some of the characteristics needed for analysis are:

- * A nested DO loop must be in the form of a tight nest, which means that no statement may occur between the DO statements and that the same statement ends all the DO loops. To obtain this, assignment statements that are located between DO statements are moved deeper into the nest. If this is not possible, new array variables are introduced or the assignment statement is eliminated and the right hand side of the assignment is substituted for all occurrences of the assigned variable within the loop.
- * The incrementation of the loop index must have a constant value, known by the compiler.

- * All assignments of the loop body must be to array elements. If this is not the case, new array variables can be introduced.
- * Each DO index must only occur in one subscript position and for all the references to a particular array it must appear in the same subscript position.

To analyse the parallelism of the loop body, a time-relative ordering of the references to the array elements is calculated.

```

      DO 10 I = 1,10,1
      DO 10 J = 1,10,1
1      A(J,I+1) = ...
2      ...      = ...A(J+2,I)
      . . .
10     CONTINUE

```

In this example we see that the reference to $A(3,2)$ on line 1 is done when $I=1$ and $J=3$, whereas the reference on line 2 is done when $I=2$ and $J=1$. The time-relative ordering of the reference on line 2 versus the reference on line 1 is represented by the 2-tuple $\langle -1, +2 \rangle$, where the first element of the tuple refers to the index variable I and the second one to J . The analyser notes that all references to a particular element of the array A in line 1 and line 2 can be represented by this same 2-tuple.

In a general case with n levels of nested loops, the ordering is represented by an n -tuple.

To see which reference occurs first, one simply looks at the sign of the first non-zero element of the ordering tuple. In our example this element is negative which indicates that the reference on line 1 to a particular element in the array occurs before the reference on line 2. When a DO loop is replaced by a DO FOR ALL statement (this is done by the synthesizer part of the Paralyzer), the element in the ordering tuple which corresponds to the index variable of the DO FOR

ALL statement is set to zero. For a DO FOR ALL synthesis to be valid, i. e. the transformed loop computes the same final value as the original FORTRAN loop, the first non-zero element of the ordering tuple must have the same sign as the first non-zero element of the original ordering tuple. This assures that the data dependencies are unchanged.

In our example this means that the nested DO loop can be replaced by a DO FOR ALL (J) (ordering tuple becomes $\langle -1, 0 \rangle$) or a DO FOR ALL (I,J) (ordering tuple becomes $\langle 0, 0 \rangle$) but not by a DO FOR ALL (I) (ordering tuple becomes $\langle 0, +2 \rangle$, which means that the first non-zero element changes sign).

Using this rule, the synthesizer tries to replace DO loops by DO FOR ALL statements. To allow the transformations, it is often required that the order of statements is changed within the loop body. By doing this, it may be necessary to introduce new temporary arrays in order that the data dependencies are kept intact.

One obvious problem with the Paralyzer is deciding which of all possible transformations should be chosen for a particular DO loop. Since the index/indices of a DO FOR ALL statement implies a certain allocation scheme for the arrays in the loop body, the choice also affects transformations of other DO loops where the same array is referenced.

6.3.3 PARALLEL PASCAL

Parallel Pascal [Ree-1, Ree-2, Ree-3] is one of the languages which have been proposed for the Massively Parallel Processor (the MPP) [Bat-5]. The language is defined in terms of extensions to Pascal. As compared to standard Pascal, the new language is extended in several ways:

Data can be declared as "parallel", which means that it should be located in the parallel processing array of the MPP. Parallel data can have one or two of its dimensions stored in parallel. In the intended applications for the MPP -image processing on two dimensional data - the elements of a declared array will be stored in the processing array so that their physical location directly corresponds to their logical position in the declared array (row, column), for example:

```
var a : parallel array[1..128,1..128] of integer;
```

Expressions can be formed with parallel arrays. If an expression contains an unsubscribed parallel array then the operation will be performed on all elements of the array. For example:

```
var a,b,c : parallel array[1..128,1..128] of integer;
```

```
a:=b+c;
```

means that $a[i,j] := b[i,j] + c[i,j]$ for all i,j in $[1..128]$.

Several standard functions which may be used with parallel arrays are included. These are defined for all sizes and shapes of arrays. Functions include Shift and Rotate of arrays any number of steps along one or several of its dimensions. Other functions are the reduction functions which are based on the primitive reduction functions in APL. Examples of these are:

asum	summation
aproduct	product
aand	logical and
amax	maximum value

If, for example, M is a two-dimensional parallel array then we obtain the following results when applying the asum function to m:

`asum(M,1)` A vector containing the sums of the
 elements in each column.
`asum(M,2)` A vector containing the sums of the
 elements in each row.
`asum(M,1,2)` A scalar containing the sum of all
 elements of M.

To specify that an operation should be performed in a subset of the processing elements of the parallel processing array, Parallel Pascal extends the meaning of the if-then-else construct in the following way: The control value may be a parallel expression which results in a parallel Boolean value. The statement following "then" is executed in those processing elements where the control value is TRUE. For example:

```

var a,b,c : parallel array[1..128,1..128] of integer;
           if a>b then c:=c+a;
  
```

Restrictions are that all statements which are contained in the body of the parallel if-statement must be conformable with the control value and may not include any assignments to scalars or any goto statements.

Finally, Parallel Pascal extends the array indexing scheme of Pascal and allows the specification of subarrays in several ways:

A selects all elements of A
 A[2,3] select one element of A as in standard Pascal
 A[2,] select a vector containing the second row of A
 A[2,4..7] select a vector containing the elements
 A[2,4], A[2,5], A[2,6]] and A[2,7]

In many applications it is useful to select a single bitposition of the elements in an array. Parallel Pascal allows a bit specification following the regular array index. For example:

AC,:0] select the sign bit position in
all elements of A

The implementation of Parallel Pascal has been done in two ways. First a Parallel Pascal Translator has been written, which translates a Parallel Pascal program into an ordinary Pascal program, i. e. all parallel statements are rewritten as loops. This is used for debugging purposes; a program can be written and tested on any conventional computer which includes a standard Pascal compiler before it is transferred to the MPP.

For the implementation on the MPP, a compiler which generates an extended form of Pascal p-code is used. P-code, or pseudo code, can be seen as the assembly language for a hypothetical stack organized computer. To execute the p-code on a "real" computer, either an interpreter which simulates the hypothetical machine is used or, since this results in a relatively inefficient solution, a code generator is used to generate the final machine code for the target computer from the p-code.

This kind of implementation, where the high level language compiler generates p-code instead of machine code for the target computer, has several advantages. The output from the compiler is in a way transportable; only the code generator must be rewritten to generate machine code for a new machine. If the p-code is carefully designed, it may also serve as output from other compilers. This means that the implementation of another high level language requires less effort, since the same code generator can be used. It is also possible to do some optimization directly on the p-code level. This optimization is independent both of the source code and of the machine code of the target computer.

The designers of Parallel Pascal have specified extensions to FORTRAN and to a subset of APL similar to the extensions of Parallel Pascal. Compilers for Parallel FORTRAN and Parallel APL are supposed to generate the same p-code as the Parallel Pascal compiler.

Chapter 7

PASCAL/L - A HIGH-LEVEL LANGUAGE FOR LUCAS

7.1 INTRODUCTION

As we have seen in Chapter 6 there are two different approaches to the use of high-level languages for parallel computers. Either the parallelism of the computer has a correspondance in the syntax of the language and special constructs are used to express parallel operations on data, or the language does not contain any primitives for parallel processing, in which case the compiler is responsible for detecting inherent parallelism in programs that are written in a sequential language and to generate code for the parallel computer.

At first glance, the second approach seems to be the more appealing of the two. The user does not have to learn a new language. Existing programs can directly be moved to the parallel computer. Programs can be developed and tested on an ordinary sequential computer before they are transported to the parallel machine. The language could also be independent of the parallel structure of any particular machine.

However, if the parallelism is not apparent in the language, the user will not be motivated to design algorithms which are suitable for parallel computation. A sequential language forces the programmer to

transform an inherently parallel algorithm into sequential code. It is also unlikely that a completely sequential algorithm could be transformed to run efficiently on a parallel machine. Thus in the interest both of efficiency and understanding of parallel algorithms we favour a language where the parallelism is visible.

For several reasons it is preferable to use an existing sequential language as a base, when defining the new language:

- * Sequential operations are indeed necessary and must be included in the language anyway.
- * The implementation may be simplified in that existing compilers can be modified to accept the new language.
- * The user needs to learn relatively few new concepts.
- * The use of the language is not restricted to parallel algorithms and the same language can be used to program the entire system including compilers and operating system.

Several different languages were considered for the choice of a suitable sequential language. APL deals with parallel arrays of data in a very general way and many of the ideas in APL are relevant to parallel processing on an SIMD computer. However, the dynamic data structures in APL and the powerful operations on these would make it very difficult to achieve an efficient implementation. FORTRAN is currently the most used language in many of the applications where LUCAS may be used. On the other hand, its poor data structures makes it very unsuitable for database management, which is one of the main application areas for LUCAS [Kru-3].

Pascal is a well structured language with powerful control and data structures which makes it suitable for many different applications. It has strong typing of variables, and a large amount of error detection is possible both at compile time and at run time.

Compilers for Pascal are relatively uncomplicated to implement. The syntax has been chosen so that only one symbol lookahead is needed, enabling the use of simple parsing techniques. To facilitate the code generation and to allow compilers to be portable, an implementation scheme with code generation for a stack oriented virtual machine is used. The fact that portable compilers - written in Pascal - exist, simplifies the implementation on different machines.

We decided that the new language, Pascal/L(UCAS) [Fer-1], should be defined in terms of extensions to Pascal. The extensions are chosen so that the new language is complete with respect to the processing capabilities of LUCAS. This means that typical SIMD operations, where one instruction operates on several data items, can be specified. A characteristic property of associative processing is the ability to designate the part of data which will be subject to parallel computations in terms of properties of the data, regardless of where it is stored.

Only constructs which can be efficiently implemented on LUCAS are included. Since the use of LUCAS is restricted to algorithms which are well suited for the architecture, this is quite reasonable. Floating point arithmetic, for example, is not included.

The following extensions to Pascal are defined:

- * Declaration of variables that will be allocated to the Associative Array. In the following these will be referred to as 'parallel variables', whereas 'scalars' or 'sequential variables' stand for variables which are located in the memory of the Host.
- * An indexing scheme to access parts of parallel variables.
- * Expressions and assignments involving parallel variables.
- * An extended control structure, allowing the use of parallel variables as control variables.

- * Standard functions for data alignment and input and output of parallel variables.

7.2 LANGUAGE DESCRIPTION

7.2.1 Declaration of Data

The one-dimensional organization of the Associative Array makes it especially suited for operations on one- and two-dimensional arrays. In principle, arrays of any dimension can be represented in LUCAS, but the natural storing scheme where adjacent array elements also are physical neighbours, will be lost. Pascal/L is therefore restricted to arrays of one or two dimensions.

Parallel variables are characterized by their dimension and their range. The number of subscripts in the declaration defines the dimension of the variable. The range can be seen as a measure of parallelism and is given by the size of the first subscript.

There are two kinds of parallel variables: selectors and parallel arrays.

7.2.1.1 Selectors

A selector defines a Boolean vector over the Processing Elements and is intended to control the parallelism of operations. (At execution time this is accomplished by setting the Tags in those PEs where the corresponding selector element has the value TRUE.)

```

<selector type> ::=
    selector[<range>] |
    selector[<range>] := <Boolean aggregate>

```

```

<range> ::=
    <constant>..<constant>

```

```

<Boolean aggregate> ::=
    <choice> => <Boolean value>

```

```

<choice> ::=
    <constant> | <constant>..<constant> |
    <constant>..<constant> step <constant>

```

```

<Boolean value> ::=
    true | false

```

For example:

```

var SEL : selector[0..99];

```

declares a selector with the range 0..99, i. e. a selector with elements in the first 100 PEs.

```

var SEL : selector[0..99]:= (0..98 step 2 => true);

```

declares a selector with the range 0..99 where all the elements with even indices are initiated to the value TRUE and all others to the value FALSE.

7.2.1.2 Parallel Arrays

A parallel array consists of a fixed number of components which are all of the same type and which are located in the Associative Array of LUCAS. Parallel arrays can be one- or two-dimensional, where the size of the first subscript in the declaration is referred to as the range of the array. It has the property that when the first index is incremented by one in an array reference, while keeping a possible second index unchanged, the new array component will be located in the PE whose index is one higher than the PE originally referenced, but, on the same address within the PE memory. This means that for any fixed value of the second array index, all components are located in a field of the Associative Array (see Figure 4.3). This definition implies that in a two-dimensional array all components of a row are located in the same PE, while the components of a column are located in different PEs.

A component of a parallel array can be of any of the following types: signed integer, unsigned integer, fixed point number, Boolean, character or string. When declaring an array with components of any of the first three types, a precision is specified in the declaration. The precision gives the number of bits used in the computations in the case of integers, and the number of bits on each side of the 'fraction mark' (binary point) in the case of fixed point numbers. The maximum length of a string component (number of characters) is given in the declaration.

```
<parallel array type> ::=
    parallel array[<range>] of <parallel type> |
    parallel array[<range>,<constant>..of <parallel type>
```

```
<parallel type> ::=
    <parallel type identifier> | <parallel standard type> |
    record <parallel field list> end
```


<parallel type identifier> ::=

 <identifier>

<parallel standard type> ::=

 integer(<constant>) | unsigned integer(<constant>) |

 fixed(<constant>.<constant>) | Boolean | char | string(<constant>)

<parallel field list> ::=

 <parallel record section> {;<parallel record section>}

<parallel record section> ::=

 <field identifier> {,<field identifier>} : <parallel standard type>

<field identifier> ::=

 <identifier>

For example:

```
var PARA : parallel array[0..99,0..2] of integer(32);
```

declares a two dimensional array, where the components are of type signed integer with a precision of 32 bits (including the sign bit). Components of the array are located in the first 100 PEs with 3 components in each PE.

```
var OBSERVATION : parallel array[1..64] of
                    record
                        SITE : string(20);
                        TEMP : fixed(8.4);
                    end;
```

declares a one dimensional array where each PE from 1 upto 64 stores a record. Each record has two fields, the first is a string with a maximum of 20 characters, the second a fixed point number where the integer part has a precision of 8 bits (including the sign bit) and the fraction part a precision of 4 bits).

In correspondence with the Pascal terminology, the word 'field' is used to denote components of a record. We have already used the same word when referring to 'fields' in the Associative Array of LUCAS. However, this should cause no confusion since in a parallel array of record, a logical field always is allocated to a physical field in the Associative Array.

7.2.2 Indexing of Parallel Variables

Indexing of a variable of array type in Pascal is used to single out one component of the array. When operating on a parallel variable, Pascal/L makes it possible to reference several components along the parallel dimension at the same time. This set of elements is referred to as the referenced range of the variable, as compared to the declared range which is specified in the declaration of the variable.

The indexing scheme allows simultaneous access to a column (or a subset of the column components) of a two dimensional array, but not to any other part of the array such as a row or a diagonal. This limitation is due to the one-dimensional organization of PEs in LUCAS, which implies that accessing any other part of an array than a column would result in a large overhead. The components of a row are all located in the same PE, which means that they must be sequentially processed. To access a diagonal, or any other part of the array where the components are located in different PEs - except a column - the PEs would have to include an index register to allow different addresses being used in different PEs.

Several ways of specifying the first index exist. If an entire column is referenced, this is denoted by a star '*' in the index position. A part of a column is referenced either by constant indices, in which case the referenced part is known at compile time, or by a selector expression. When a selector expression is used, the array components referenced are identified by the indices where the expression takes the value TRUE. A one-dimensional parallel array may be used without

any index at all (and no brackets), in which case all components of the array are referenced.

```
<indexed parallel variable> ::=
    <parallel variable identifier> |
    <parallel variable identifier>[ <first index>] |
    <parallel variable identifier>[ <first index>,<expression>]
```

```
<parallel variable identifier> ::=
    <identifier>
```

```
<first index> ::=
    * | <constant> | <constant>..<constant> | <selector expression>
```

Examples:

P0	Select all the elements of the one-dimensional variable.
P1[* ,0]	Select column 0 of P1. P1 is a two-dimensional parallel variable.
P1[S,0]	Where S is a selector: select a subset of column 0 of P1
P1[2..80,0]	Select a subset of column 0 of P1.

7.2.3 Expressions and Assignments

It is possible to combine sequential and parallel variables in expressions as long as no type conflict occurs. This means for example that it is allowed to form expressions where a scalar integer is combined with a parallel array of integers.

An expression or part thereof which only contains sequential variables and constants results in a sequential value. An expression

or part thereof which includes at least one parallel variable results in a parallel value, unless the parallel variable(s) is used as a parameter to a function which yields a scalar result.

In the computation of a parallel expression, all referenced ranges to parallel variables must be identical and any sequential value is parallelized to this range before evaluation of the expression. This means that

```
4+PARA[*]
```

results in 4 being added to all the components of PARA and

```
4+PARA[2..5]
```

results in 4 being added to components 2,3,4 and 5 of PARA.

There are four kinds of assignment statements:

- 1) The left hand side and the right hand side are both scalars. This is the normal Pascal assignment statement
- 2) The left hand side is a parallel variable and the right hand side is a sequential value (expression). In this case all the components within the referenced range of the parallel variable are assigned the value of the scalar expression.
- 3) The left hand side is a sequential variable and the right hand side is a parallel expression. The referenced range of the parallel variables must be such that the value of the right hand side expression includes one single component.
- 4) The left hand side is a parallel variable and the right hand side is a parallel expression. The referenced components of the left hand side variable are assigned the corresponding elements of the right hand side expression. The range of the expression must be equal to, or overlap, the referenced range of the left hand side variable.

The following program exemplifies different kind of assignments.

Program Assign;

var ODD : selector[0..127]:=(1..127 step 2 => true);

EVEN,

SEL : selector[0..127];

P1,P2 : parallel array[0..127] of integer(16);

I : integer;

begin

...

EVEN:=not ODD; { Both sides parallel. Same range }

P1[EVEN]:=P2*2; { Both sides parallel. The range
{ of the righthand side expression
{ overlaps the referenced range of P1 }

P1[ODD]:=0; { Left hand side parallel, right hand
{ side scalar. All the odd elements of
{ P1 are assigned the value zero }

I:=P2[5]; { Left hand side scalar, right hand
{ side parallel, but the referenced
{ range includes one single element }

SEL:=P1 > P2; { Both sides parallel. Same range }

I:=P2[SEL]; { Left hand side is scalar, right hand
{ side is parallel. SEL must have one
{ single component with the value TRUE }

end.

7.2.4 Control Structure

Pascal contains five structured constructs which control the sequential program flow: the if, case, while, repeat and for statements. The first two are used to select different paths in the program execution, while the remaining three control the repetition of statements.

In a sequential Pascal program, all the actions taken can be ordered according to the time interval in which they occur. This ordering defines the program flow and is directed by the control statements and by the order in which statements are written in the program. The execution of programs on LUCAS differs from this in that as many as 128 actions may occur during the same interval, each in a different Processing Element. In the same way as the control constructs in Pascal determine the execution along the time dimension, new concepts are included in Pascal/L to allow the control of selection and repetition along the parallel dimension.

The construct:

```
if Boolean expression then true-statement  
                                else false-statement
```

in Pascal selects one of two different paths in the program flow, depending on the value of the Boolean expression. In the corresponding parallel statement, the Boolean expression yields a selector. Elements of the selector determine if the true-statement or the false-statement will be executed on the corresponding data elements.

In a global perspective this means that both paths will be followed and that both the true-statement and the false-statement are executed, but on different data and in different PEs. Rather than to extend the meaning of the if-then-else construct, we define a parallel selection with the following form:


```

where selector expression do true-statement
                                elsewhere false-statement

```

where the elsewhere-part is optional.

Analogous to the Pascal case statement, which is a generalization of the if-then-else construct, where the selection of one out of many statements is based on the value of the controlling expression given at the head of the case -statement, Pascal/L defines a parallel form of the case-statement. In accordance to the where-do-elsewhere construct, the parallel case does not result in one execution path being followed, but all, each working on different data. The form of the parallel case statement is:

```
case where parallel expression of
    constant : statement;
    constant : statement;
    . . .
    constant : statement;
    others : statement;
end;
```

where the others-part is optional. Like in Pascal, statements may be of the form compound statements, i. e. a list of statements, separated by semicolons, preceded by the keyword begin and followed by the keyword end, is regarded as one single statement and may appear at any place where a statement appears.

In a similar way an extension to the Pascal

```
while Boolean expression do statement
```

is defined to control repetition for parallel data:

while and where selector expression do statement

Here the statement is repeated as long as the selector expression takes the value TRUE in any element. However, during each repetition of the statement, the selector expression also decides in which PEs the statement should be executed.

The following example shows how $V[I] \bmod N[I]$ can be calculated for every element of the two vectors V and N using repeated subtractions:

```
var V,N : parallel array[0..127] of integer(16);
      ...
  while and where V >= N do V:=V-N;
```

7.2.5 Standard Functions and Procedures

7.2.5.1 Data Alignment

In expressions and assignments where the components of the parallel variables are located in different PEs, the variables must be aligned. The kind of alignment needed is defined by the programmer in terms of standard functions, which correspond to the possible data movements over the communication network in LUCAS.

```
shift(<parallel variable identifier>,I)
rotate(<parallel variable identifier>,I)
```

The first of these functions shifts a parallel variable (parallel array or selector) I steps along its first dimension, placing component N in position N+I. Zero-elements are shifted in from the edge. This corresponds to moving data up or down in the Associative Array. The rotate function is similar to the shift function except that the elements that are shifted out at one edge are shifted in at the opposite edge of the parallel variable.

```
shuffle(<parallel variable identifier>)
exshuffle(<parallel variable identifier>)
```

The elements of the parallel variable (parallel array or selector) are permuted according to the Perfect Shuffle/Exchange network on LUCAS. These functions are only defined for variables with the declared range 0..127. The first function performs a shuffle of the elements, placing component N with index $n_0n_1n_2\dots n_k$ in position $\text{Shuffle}(N)$ with index $n_1n_2\dots n_kn_0$ (see Section 1.6.2). The second function performs a shuffle followed by a pairwise exchange of the elements, placing component N with index $n_0n_1n_2\dots n_k$ in position $\text{Exshuffle}(N)$ with index $n_1n_2\dots n_kn_0'$, where n_0' denotes that the last bit in the index is complemented.

7.2.5.2 Selector Operations

```
first(<selector expression>)
```

This function is used to find the first component of a selector expression with the value TRUE. It returns a new selector with only this element TRUE.

```
next(<selector identifier>)
```

The next-procedure assigns the value FALSE to the first true element of the parameter, which must be a variable. This is useful in associative processing, when the result of a parallel search operation is processed sequentially.

```
any
```

The any-function returns the value FALSE if a previous call to the first-function or the next-procedure returned an all-false selector,

otherwise it returns the value TRUE.

some(<selector expression>)

A call to the some-function evaluates the selector expression and returns the value TRUE if it contained at least one TRUE element, otherwise the value FALSE is returned.

Example:

```
var PAR1 : parallel array[0..99] of unsigned integer(12);
    SEL  : selector[0..99];
    SUM  : integer;
begin
    SUM:=0;
    SEL:=PAR1 > 10;
    while some(SEL) do
        begin
            SUM:=SUM+PAR1[first(SEL)];
            next(SEL);
        end;
    end;
```

In the example SUM gets the sum of all the elements of PAR1 whose value is greater than 10.

7.2.5.3 Input and Output

The Pascal standard procedures read and write must be extended to allow input and output of parallel variables. Details of how this should best be accomplished have not been worked out. As a preliminary attempt the procedures are extended so that they may take

variables denoting parallel arrays as parameters, meaning that whole parallel arrays may be read or written.

7.2.6 Microprograms

It is possible to explicitly invoke a microprogram which has been written in the microprogramming language. This allows a significant speedup for parts of the program that can be expressed in microcode, i. e. parts which only include parallel operations. Examples of such operations are matrix multiplications and image operations.

The microprogram must be declared in the declaration part of the program. The syntax of the declaration is similar to the syntax of microprogram headings in the microprogramming language:

```

<microprogram declaration> ::=
    microprogram <identifier> <microprogram parameter list> ; external;

<microprogram parameter list> ::=
    <empty> |
    <microprogram parameter>{,<microprogram parameter>}

<microprogram parameter> ::=
    <empty> | <identifier>
  
```

A standard function which is used in conjunction with microprograms is the following:

```
location(<parallel variable identifier>)
```

This function results in an integer value, which indicates the bit address to the least significant bit of the parallel variable in the Associative Array.

Invocation of microprograms is similar to procedure calls.

```

var  M1,M2,M3 : parallel array[0..127,0..127] of integer(16);
Microprogram Matmult(A,B,C,precision); external;

begin
    ....
    Matmult(Location(M1),Location(M2),Location(M3),16);
    ....
end;

```

7.3 EXECUTION ON LUCAS

To answer the questions of what is meant by a certain construct and how it should be executed on a target computer, semantic rules must accompany every syntactic construct. The extensions defined by Pascal/L have been chosen in the spirit of Pascal, where the syntax reflects the underlying semantic rules. In this section we will look at some of the constructs in Pascal/L and see how they should be interpreted.

A convenient way to define the semantics of a language is to show how its constructs are translated into code for some model of a machine. In the case of Pascal, a virtual stack-oriented machine is often used as the target computer for the generation of intermediate code by the language compiler. In order to clarify the semantics of a construct, the code which is generated for this 'pseudo-machine' when the construct is compiled can be given.

In this section we will define an extension to the Pascal pseudo-machine and present a part of the instruction list which is adequate to describe the execution of two important constructs in

Pascal/L: parallel expressions and the where-do-elsewhere statement.

7.3.1 Pascal/L Pseudo-machine

In the description of the machine we will concentrate on the parts necessary for the evaluation of parallel integer expressions and we will deliberately leave out details about procedure entry and exit (the construction of stack frames), traversal of links in order to access non-local variables etc. The emulation of the pseudo-machine on LUCAS is straight-forward and will not be discussed.

The Pascal/L pseudo-machine, or p-machine, has several registers and uses three distinct memory areas as shown in Figure 7.1. In an implementation of the p-machine on LUCAS, two of the memories would be located in the memory area of the Host (the Program Memory, the Stack) and the third in the Associative Array of LUCAS (the Parallel Memory).

The Program Memory holds the instructions of the program being executed. A register, PC, points to the instruction that will be executed next.

The Stack contains sequential variables, sequential temporaries and pointers to locations in the Stack, the Program Memory and the Parallel Memory. Two registers point to locations in the Stack: SP, the stack pointer, points to the top-of-stack element and AP, the activation pointer, to the activation record of the currently executing procedure.

The Parallel Memory holds parallel variables and parallel temporaries. Each entry in the Parallel Memory is a 128-element vector and is defined by a descriptor, which is located in the Stack. A descriptor consists of a pointer to the Parallel Memory and a format specification giving the precision and the type of the variable. The Parallel Memory is organized in the form of a stack, the Parallel Stack, where each entry has 128 values. The register PSP points to

the top element of this stack. The parallel stack contains parallel variables as well as parallel temporaries, which are used during expression evaluation to hold the intermediate results.

Each parallel variable has an associated bit-slice, the range indicator, which indicates the declared range of the variable. A temporary on the Parallel Stack has a corresponding range indicator giving its actual range.

The register RP points to a bit-slice in the Parallel Memory where the Current (evaluation) Range is stored. This range is set either when executing a parallel control statement, e. g. the where-do-elsewhere statement, or as the result of an indexing operation.

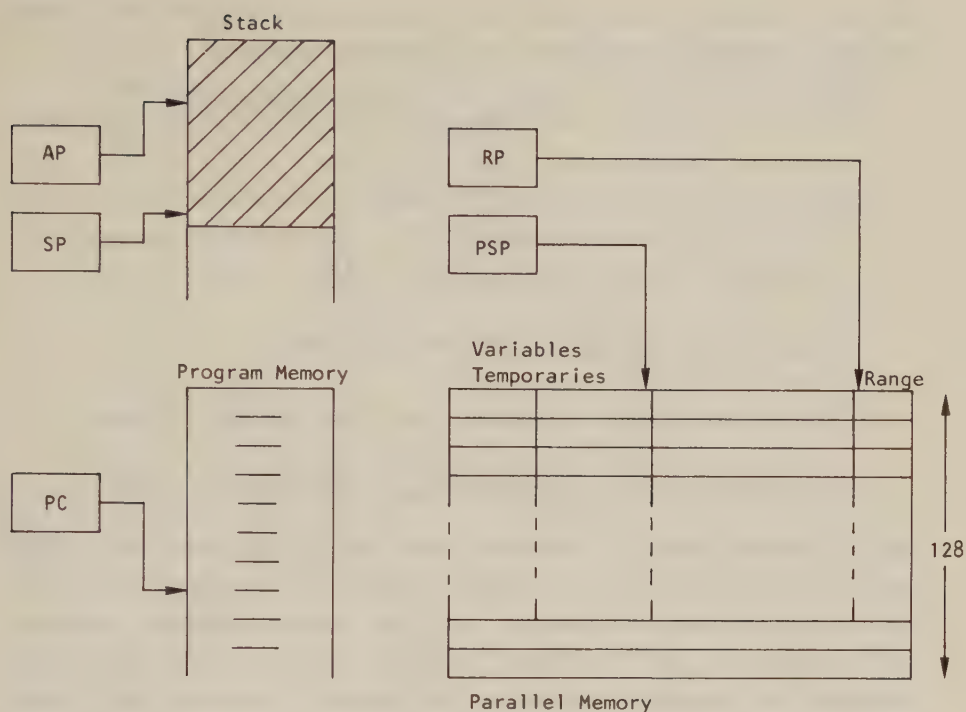


Figure 7.1 The Pascal/L pseudo-machine

The Stack is essential for the evaluation of expressions and is used to reference all the operands. In stack-oriented machines all operands are pushed onto the stack from where they are removed by the arithmetic and logical operators. Once the operation is completed the result is pushed back onto the stack. When operating on parallel variables in the Pascal/L p-machine, it is often enough to push a descriptor on the Stack without also pushing the variable itself onto the Parallel Stack. In many simple expressions (like adding two

parallel variables and storing the result in a third) this will result in a considerable reduction of the overhead involved.

The value of an entry in the Stack may have several interpretations:

- * a scalar value
- * a pointer to another entry in the Stack
- * a pointer to a location in the Program Memory
- * a descriptor to a variable in the Parallel Memory
- * a descriptor to a temporary on the Parallel Stack.

Upon procedure entry, a local data area for the procedure is created both in the Stack and in the Parallel Memory. On the Stack this takes the form of a reserved memory area for scalar variables, together with descriptors to the local parallel variables. The bit-slices indicating the declared ranges of the parallel variables are loaded when space is reserved in the Parallel Memory. Selector variables are initiated if needed.

The instructions needed to demonstrate expression evaluation will now be described. In the following, TOS stands for the entry on top of the Stack and OP for the Stack entry which is addressed by the operand field of an instruction.

LOAD type,lev,disp

This instruction puts a variable on top of the Stack. The location of the variable depends on the values of 'lev' and 'disp'. Lev indicates the number of static levels to traverse in order to find the activation record and disp is the offset within the activation record to find the variable. Together they define the OP entry in the Stack. Depending on the value of the type-parameter, different actions are

taken:

- 0 Load a scalar. The value of the scalar is pushed on the Stack
 - SP is incremented and the value is put in the location
 indicated by the new value of SP.

- 1 Load a parallel variable. The descriptor of the variable is
 pushed on the Stack, but the variable is not moved to the
 Parallel Stack.

LIT value

This instruction loads the literal specified in the parameter on the Stack.

COPY type

Push TOS onto the Stack, i. e. make a duplicate of the element on top of the Stack. Depending on the type-parameter the following actions may be taken:

- 0 Copy a scalar. The value of TOS is pushed on the Stack.

- 1 Copy a parallel variable. The descriptor which is in TOS is
 pushed on the Stack, but the variable is not moved to the
 Parallel Stack.

- 2 Copy a parallel temporary. The descriptor which is in TOS is
 pushed on the Stack, a copy is made in the Parallel Stack.

STORE type,lev,disp

This instruction stores TOS in the OP location. Depending on the value of the type-parameter, the following actions may be taken:

- 0 TOS and OP are both scalars. Copy TOS into location OP and pop TOS.
- 1 TOS and OP are both parallel variables. They are both represented on the Stack by descriptors to entries in the Parallel Memory. First compute a selector by performing the operation AND between the declared range of OP and the Current Range - as indicated by register RP. Then use this selector while performing a field copy operation in the Parallel Memory. Check that the declared range of TOS overlaps the selector, i. e. that TOS is defined in every component which has been copied, and if not, raise a run time error. This check operation will be described below. Pop the TOS descriptor off the Stack.
- 2 TOS is a parallel temporary and OP is a parallel variable. They are both represented on the Stack by descriptors to entries in the Parallel Memory. First compute a selector by performing the operation AND between the range indicator of OP and Current Range - as indicated by register RP. Then use this selector while performing a field copy operation in the Parallel Memory. Check range overlapping as described below. Pop TOS off the Parallel Stack by adjusting PSP and pop the TOS descriptor off the Stack.
- 3 TOS is scalar and OP is a parallel variable. Compute a selector as above and use it while performing a field load in the Parallel Memory. Pop TOS.
- 4 TOS is a parallel variable and OP is a scalar. Compute a selector as above. Perform a check that this selector has one single TRUE element, and if not, raise a run time error. Use the selector to read out the variable element and store it in

the OP location. Pop the TOS descriptor off the Stack.

- 5 TOS is a parallel temporary and OP is a scalar. Compute a selector as above. Perform a check that this selector has one single TRUE element, and if not, raise a run time error. Use the selector to read out the variable element and store it in the OP location. Pop TOS off the Parallel Stack by adjusting PSP and pop the TOS descriptor off the Stack.

When checking that the declared range of TOS overlaps the selector, S, which is used for the field copy operation above, the implication

$$\text{TOS.declared range}[i] \Rightarrow \text{S}[i]$$

must hold for a all $i: 0 \leq i < 127$. On LUCAS this may be efficiently executed in the Associative Array (using the Some/None capability).

STIN type

This store-indirect instruction is similar to STORE, but the target address is in the second element of the Stack (TOS-1), and not specified in the instruction. The type-parameter has the same meaning as in the STORE instruction.

NOT/NEG type

These are unary instructions for forming the Boolean complement and the arithmetic negation. They operate on TOS, pop the Stack and push the result back on the Stack. Depending on the value of the type-parameter we have:

- 0 TOS is a scalar. Perform the operation and replace TOS with the result.
- 1 TOS is a parallel variable. It is represented on the Stack by a

descriptor to an entry in the Parallel Memory. Execute the instruction on the entry in the Parallel Memory which is described by TOS. Put the result in the Parallel Stack with the the declared range of TOS stored as range indicator, adjust PSP accordingly and update TOS to indicate that the top element now is a temporary.

- 2 TOS is a parallel temporary. It is represented on the Stack by a descriptor to an entry in the Parallel Memory. Execute the instruction on the entry in the Parallel Memory which is described by TOS. Leave the result in the same location of the Parallel Memory without changing the range indicator.

ADD/SUB/MULT/DIV/MOD type

These instructions represent binary operations which operate on the two top elements of the Stack. Similar to the previous instructions, the type-parameter indicates if both operands are scalars or if one or both operands are parallel. In the case where both are scalars, the result is pushed onto the Stack after the two operands have been popped. If at least one of the operands is parallel, then the operation is performed in the Parallel Memory, leaving the result on the Parallel Stack after the operands have been popped.

The type-parameter can take any of the values 0 to 8, as seen in Table 7.1.

TOS-1 is
=====

	scalar		parallel variable		parallel temporary	
TOS is =====	-----					
scalar		0		1		2
parallel						
variable		3		4		5
parallel						
temporary		6		7		8

Table 7.1 The value of the type-parameter in binary operations

SETRANGE type

Compute a new value of Current Range by performing a Boolean AND between Current Range and the selector whose descriptor is TOS. Push the old value of Current Range onto the range stack by adjusting RP. Depending on the value of the type-parameter, the following actions are also taken:

- 0 TOS is a parallel variable. Pop the TOS descriptor off the Stack.
- 1 TOS is a parallel temporary. Pop TOS off the Parallel Stack by adjusting PSP and pop the TOS descriptor off the Stack.

POPRANGE

Restore Current Range to a previous value by popping the range stack.

7.3.2 Parallel Expressions

In order to use the p-machine for evaluation of a parallel expression in a Pascal/L program, the expression is translated by the compiler to a form of postfix notation. This form is ideal when a stack is used to compute the expression, since the following simple rule may be used while scanning the expression from left to right:

If the next symbol is an operand then push its value on the stack, else (it is an operation) use the element(s) on top of the stack as operand(s) to the operation, pop the operand(s) off the stack and push the result.

When starting the computation, the stack is empty and when the end of the expression is reached, the result is the only element left on the stack. Since we deal with variables as well as constants, the notation is extended so that an operand no longer is represented by its value but by an instruction which should be executed in order to get the value on top of the stack.

This describes a commonly used technique for the intermediate code in language compilers and is the philosophy behind the Pascal p-code [Wir-1]. The code generated from a Pascal/L compiler would consist of instructions similar to those described in Section 7.3.1.

Without dealing with how the transformation process works, we will look at an example of a parallel assignment statement in Pascal/L.

```

var P1,P2 : parallel array[0..127] of integer(32);
    ODD    : selector[0..127];
    I      : integer;
begin
    ...

    P1[ODD]:=P2-P1*(2+I);

```

The statement $P1[ODD] := P2 - P1 * (2 + I)$ has been translated into parallel p-code and will be executed on the Pascal/L p-machine. In the p-code program, which is shown in Figure 7.2, we have replaced the lev/disp-parameters with the name of the variable they refer to.

	Instr.	Parameter	TOS becomes
1	LOAD	1,P1	descriptor to par. var. P1
2	LOAD	1,ODD	descriptor to selector ODD
3	SETRANGE	0	descriptor to par. var. P1
4	LOAD	1,P2	descriptor to par. var. P2
5	LOAD	1,P1	descriptor to par. var. P1
6	LIT	2	scalar (the value 2)
7	LOAD	0,I	scalar (value of I)
8	ADD	0	scalar (value of 2+I)
9	MULT	1	descriptor to par. temp.
10	SUB	7	descriptor to par. temp.
11	STIN	2	<empty>
12	POPRANGE		<empty>

Figure 7.2 Parallel p-code for the statement $P1[ODD] := P2 - P1 * (2 + I)$

7.3.3 Where Statement

The general idea for executing a control statement of the form:

```

where selector expression do statement-a
                                elsewhere statement-b
  
```

on the Pascal/L p-machine is that the selector expression is used to calculate two new values of Current Range which are used when executing statement-a and statement-b respectively. By using a stack in the Parallel Memory to save the old value of Current Range, the problem of how to handle nested where-statements (and similar constructs) is solved. Upon entry of a where-statement, Current Range is pushed on the stack and restored after the statement has been executed.

A first attempt to translate the where-do-elsewhere construct results in the p-code given in Figure 7.3.

Instr.	Parameter	TOS becomes
0	selector expression	resulting selector (var. or temp.)
1	COPY	1 or 2
2	NOT	1 or 2
3	SETRANGE	1
4	SETRANGE	0 or 1
5	statement-a	<empty>
6	POPRANGE	<empty>
7	statement-b	<empty>
8	POPRANGE	<empty>

Figure 7.3 Preliminary translation of the where-construct

Before executing the instruction on line 1, we assume that the selector expression is in TOS. Depending on the type of expression,

this is either a parallel variable or a parallel temporary. The first instruction produces a copy of the value on the Stack. Instructions 2 and 3 invert the value and calculate a new Current Range by masking the present Current Range, which is pushed onto the range stack. This is the Current Range which will be used during execution of statement-b. A second SETRANGE operation calculates Current Range for statement-a. After execution of statement-a, the range stack is popped, setting Current Range to the previous calculated value for statement-b. Execution of the where-statement terminates with restoring Current Range to its initial value.

In Section 7.2.4 where the control statements of Pascal/L were introduced, we did not discuss the semantic aspects of executing them on an SIMD computer. Intuitively we feel that the execution of the do-part and the elsewhere-part of a where-statement ought to be independent and that no order should exist between statement-a and statement-b. However, when executed on LUCAS as described above, the two statements are processed one after the other. The following example illustrates why the scheme presented is insufficient to assure that the result corresponds to the desired semantics of the construct.

```
var  P1,P2 : parallel array[0..3] of integer(32);
      ODD   : selector[0..3]:=(1,3 => true);
begin
      ...
      where ODD do P1:=rotate(P2,1)
           elsewhere P2:=rotate(P1,1);
```

Assume that the initial values are as shown in Figure 7.4 (a). Depending on the order of execution of the statements, the result will be different as seen in the figure. If we decide that the result of executing one statement, say statement-b, should be independent of whether statement-a has been executed or not, we must require that statement-a does not change any variables until statement-b has been

executed. And similarly the other way around.

Figure 7.4 (b) shows the result in this case, where both statements calculate their results and update the variables when both are terminated. Note that these updates are independent of the order of execution since they use different values on the Current Range. Figure 7.4 (c) shows the result when statement-a is executed before statement-b and Figure 7.4 (d) when statement-b is executed before statement-a.

index	P1	P2	P1	P2	P1	P2	P1	P2
0	a	A	a	d	a	C	a	d
1	b	B	A	B	A	B	d	B
2	c	C	c	b	c	A	c	b
3	d	D	C	D	C	D	b	D
	a)		b)		c)		d)	

Figure 7.4 Result depends on the order of execution. (a) initial values. (b) independent execution. (c) statement-a executed before statement-b. (d) statement-b executed before statement-a

In order to obtain independence between the two statements, temporary locations must be used to store the new values of assigned parallel variables, resulting in the p-code shown in Figure 7.5.

Instr.	Parameter	TOS becomes
0	selector expression	resulting selector (var. or temp.)
1	COPY	second copy
2	NOT	inverted second copy
3	SETRANGE	1
4	SETRANGE	0 or 1
5	statement-a (modified)	<empty>
6	POPRANGE	<empty>
7	statement-b	<empty>
8	POPRANGE	<empty>
9	copy temporary to parallel variables	

Figure 7.5 Translation of the where-construct

While executing statement-a, all parallel variables which appear on the left hand side of an assignment (statement-a may be a compound statement) are copied to a temporary area. For each variable copied there is also an "modify-selector", which will later be used when updating the variable. This selector is initiated to an all-false value.

Statement-a is now executed, but the following modifications have been made in the code:

- * All assignments are to the temporary locations of the variables. The corresponding modify-selector is updated to reflect which elements have been changed in the temporary location.
- * When using a variable which have been copied to a temporary location, its value is taken from this location, not from the variable.

Statement-b is executed with no changes. Finally variables are updated from their temporary locations, using the corresponding modify-selectors as indices in the updates.

A similar technique may be employed for the parallel case-statement.

7.4 PROGRAMMING EXAMPLES

Example 1. Matrix Multiplication

The first example shows a program for matrix multiplication. This program computes the product of two 128×128 matrices using the "middle-product method", as described in [Hoc-1, Sve-3]. Using this method, the inner-products are developed column-wise so that the k :th column of the result matrix is computed by multiplying each column of the first matrix with the k :th column of the second matrix while accumulating the results.

```

Program MiddleProduct;
var A,B,C : parallel array[0..127,0..127] of integer(16);
    ROW : selector[0..127] := (0 => True);
    ACOL,BCOL : integer;

begin
    for BCOL:=0 to 127 do C[* ,BCOL]:=0; (* clear C *)
    for BCOL:=0 to 127 do      (* for each column of B *)
        for ACOL:=0 to 127 do (* multiply with all columns of A *)
            begin
                C[* ,BCOL]:=C[* ,BCOL] + A[* ,ACOL]*B[ROW,BCOL];
                ROW:=rotate(ROW,1); (* and accumulate *)
            end;
        end;
    end.

```

Example 2. Outer Perimeter of Objects

An algorithm for finding the outer perimeter of objects in a binary image works as follows [Sve-3]:

- 1) Mark one picture element at the edge which belongs to the background
- 2) Propagate the marker to neighbours in the edge column which belong to the background
- 3) Copy markers to elements in the next column which belong to the background
- 4) Propagate the markers to neighbours in the column which belong to the background
- 5) Scan back and forth over the entire image until no new markers are produced

This algorithm can be expressed in Pascal/L as follows:

```

Program Perimeter;
{ Find outer perimeter of objects in Boolean image I }
var I,M : parallel array[0..127,0..127] of Boolean;
    finished : Boolean;
    k : integer;          {used to indicate columns}

begin
    { edge column }
    while and where not(MC*,0J) and not(IC*,0J) and
        (shift(MC*,0J,1) or shift(MC*,0J,-1)) do MC*,0J:=true;

    finished:=false;
    while not finished do
    begin
        finished:=true;
        { scan left }
        for k:=1 to 127 do
        begin
            MC*,kJ:=not(IC*,kJ) and MC*,k-1J;
            while and where not(MC*,kJ) and not(IC*,kJ) and

```



```

                                (shift(MC*,k],1) or shift(MC*,k],[-1)) do
begin
    MC*,k]:=true; finished:=false;
end;
end;

{ scan right }
if not finished then
    for k:=126 downto 0 do
        begin
            MC*,k]:=not(IC*,k]) and MC*,k+1];
            while and where not(MC*,k]) and not(IC*,k]) and
                (shift(MC*,k],1) or shift(MC*,k],[-1)) do
                begin
                    MC*,k]:=true; finished:=false;
                end;
            end;
        end;
    end;
end.

```

Example 3. Shortest Path

To solve the problem of finding the shortest path between all pairs of vertices in a weighted directed graph, an algorithm due to Floyd [Flo-1] is used [Sve-3]. The graph is described by a distance matrix, which gives the weight of all vertices. The absence of a direct path between a pair of vertices is represented by an "infinite" value. Starting with the original $n \times n$ matrix, new matrices are formed sequentially. The matrix produced in iteration k , D_k , is obtained from D_{k-1} by inserting vertex k in a path whenever this results in a shorter path.

```

Program Floyd;
var Dmatrix : parallel array[0..127,0..127] of integer(8);
    k,p ; integer;

begin
    for k:=0 to 127 do
        for p:=0 to 127 do
            where (Dmatrix[*,k]+Dmatrix[k,p]) < Dmatrix[*,p] do
                Dmatrix[*,p]:=Dmatrix[*,k]+Dmatrix[k,p];
            end.
        end.
    end.

```

Example 4. Project

One commonly used operation in the relational data base model is the PROJECT operation:

PROJECT R1 OVER A GIVING R2

where R1 and R2 are relations and A an attribute of R1. This operation creates a new relation, R2, from R1 by discarding attributes other than A. After that, all redundant tuples are removed from R2.

Each relation has a corresponding mark selector which indicates where tuples are defined. A description of the operation can be found in [Kru-1].

Program Project;

[illegible]

Example 5. Fast Fourier Transform

This example shows a procedure which computes the Fast Fourier Transform. Descriptions of the implementation of the FFT algorithm on LUCAS is found in [Fer-2, Sve-3]

Procedure FFT;

```
const NOOFITERATIONS = 7;
      NOOFBITS       = 8;
```

```
type COMPLEX = record RE,IM : fixed(1.NOOFBITS) end;
```

```
var X      : parallel array [0..127] of COMPLEX;
    { The complex constants reside in the global variable:
      OMEGA : parallel array [0..127,1.NOOFITERATIONS] of COMPLEX; }
    SQUARE: parallel array [0..127] of COMPLEX;
    RESULT: parallel array [0..127] of fixed(1.NOOFBITS);
    I      : integer;
```

Procedure FFTiteration(I : integer);

```
var ODD : selector [0..127] := (1..127 step 2 => True);
    MERGE1, MERGE2, MERGE3, MUL1, MUL2, MUL,
    XUPPER,XLOWER : parallel array [0..127] of fixed(1.NOOFBITS);
```

begin

```
  where ODD do MERGE1:=exshuffle(X.RE)
                elsewhere MERGE1:=shuffle(X.IM);
  MUL1 := MERGE1*OMEGAC[I].RE;
```

```
  where ODD do MERGE2:=exshuffle(X.IM)
                elsewhere MERGE2:=shuffle(X.RE);
  MUL2 := MERGE2*OMEGAC[I].IM;
```

```
  where ODD do MUL:=MUL1-MUL2
                elsewhere MUL:=MUL1+MUL2;
```

```
  where ODD do MERGE3:=shuffle(X.IM)
                elsewhere MERGE3:=exshuffle(X.RE);
```

```
  XUPPER := MERGE3+MUL;
  XLOWER := MERGE3-MUL;
```

```

where ODD do X.RE:=below(XUPPER)
           elsewhere X.RE:=XLOWER;

where ODD do X.IM:=XUPPER
           elsewhere X.IM:=above(XLOWER);

end; {FFTitration}

begin
  ...
  {sample values are input to X.RE}
  X.IM := 0;

  for I := 1 to NOOFITERATIONS do FFTitration(I);
  SQUARE.RE := X.RE * X.RE;
  SQUARE.IM := X.IM * X.IM;
  RESULT := SQUARE.RE+SQUARE.IM;
  {power spectrum is now in array RESULT}

end; {FFT}

```

7.5 CONCLUSIONS

Even though no compiler for Pascal/L has been implemented, the possibility to have a well defined language at hand for expressing parallel algorithms has proved very useful. A program in Pascal/L can be "mentally compiled" either directly to microcode or, more often, to a machine level program, which is then implemented in assembly language or in standard Pascal as described in Chapter 3. The fact that Pascal/L, although general in its structure, is well adjusted to the capabilities of LUCAS is of course important since this allows programs to be neatly and efficiently mapped onto the machine during this process.

Other languages, which are also based on Pascal, have been defined

for ILLIAC IV (Actus [Per-1]) and for the MPP (Parallel Pascal, see Section 6.3.3). In correspondence to Pascal/L they comprise facilities for declaring and operating on parallel data. However, they do also include constructs which are not directly applicable to LUCAS. On the other hand, Pascal/L defines important constructs which are not present in these languages.

A significant concept in Pascal/L is the selector, which is used to control the parallelism of operations. The same effect can of course be obtained by using the where-statement, but this leads to a less efficient result in that the condition of the where-statement must be re-computed every time the statement is executed, whereas a selector may be computed in advance. A selector is represented by a bit-slice in the Associative Array which means that it can be handled efficiently during execution. Parallel Pascal has no correspondence to the selector. Actus defines "index sets", which are used in a way similar to selectors. What limits the usefulness of these index sets is that they can only be assigned values either in the declaration or by set operations upon other index sets (union, intersection, difference and complement). The Pascal/L standard function "first", which is used to define a unique selection, has no correspondence in the other languages. This function is essential in database operations.

The definition of a Pascal/L pseudo-machine suggests how the language could be implemented on LUCAS. It would also be possible to emulate this pseudo-machine on any standard computer, which means that programs could be tested before they were moved to LUCAS. These tests could include relevant performance estimations and extensive error-checking on the p-code level, two facts that makes this approach far more appealing than the translation scheme which has been proposed for other languages, where the parallel language is translated to a standard language e. g. Pascal.

Part 5. Conclusions

Chapter 8 CONCLUSIONS

8.1 GENERAL

New forms of computer organization are most often introduced in order to meet the demands of specific applications, where the performance of existing machines is insufficient. In the case of array processors, several slightly different design concepts have emerged from the computational requirements in areas such as digital signal processing, matrix computation, image processing, data base processing and weather data processing. All these areas have one thing in common: the computations involve large quantities of well structured data which is uniformly processed. This makes an SIMD organization possible.

Based on the concept of an SIMD organized array computer, one goal of our project was to design a machine which would give a considerable improvement in performance over medium size computers in the application areas that our initial studies had led us to: signal processing and low-level image processing. At the same time we felt that our design should be general enough to support other possible applications, e. g. matrix computations or relaxation problems.

The project should be accomplished with the limited resources of a university environment in a period of five years. Three persons were involved in the project, which included everything from the initial system design to the development of adequate system software, not to mention the tremendous "overhead" in the form of wire-wrapping the prototype boards and debugging the hardware. At the same time we considered this to be only one part of our research, with further studies of applications and programming methodology being other important parts.

This thesis is divided into three main parts which cover the design of LUCAS, the design and the implementation of a microprogramming language and finally the proposal of a high-level language suitable for parallel processing on LUCAS. In this chapter we will summarize our results and discuss possible subjects of future research.

8.2 LUCAS PROCESSOR ARRAY

8.2.1 Design

The first important choice to make in the design of an SIMD computer is to decide upon the complexity of the processing elements. This choice will directly influence other design parameters, especially the size of the processor array. The question is: is it better to have a large array of very simple PEs than to have a smaller array of more complex PEs? The answer is: it depends on the application.

However, an argument which supports the former approach is that a smaller array will more often lead to situations where the data does not fit into the array and where the computation must be split up into parts. This will often result in difficulties with the boarder elements, which will have to be taken care of separately. Every such situation, which reduces the parallelism of the computation, will have a disastrous influence on performance (cf. Figure 1.2).

We decided to use simple, bit-serial processing elements and we fixed the size of the processor array to 256 PEs. This number was later reduced to 128 during the implementation phase. The bit-serial working mode has proved to be both flexible and efficient and in many applications, e. g. image processing, parallelism on the bit level would not give any significant increase in performance.

The next choice concerned the internal organization of the PEs. The number of registers was set to four, since this would assure a reasonable efficiency for arithmetic processing (result bit, carry bit, activation control and a fourth register to serve as temporary register). The implementation of the ALU in the form of a PROM is a rather expensive solution, but provides a high degree of flexibility.

Among the functions which have been considered if a more powerful design of the PEs would be possible, e. g. in a VLSI implementation, are an index register to allow a more flexible access scheme than the bit-slice, a counter which could be incremented or decremented in one clock cycle and which could be used to "switch off" the PE by resetting the Tag when the counter overflows. In many cases it would also be advantageous to have a shift register in the PE. This could be used both to hold partial products during multiplication and to align or normalize floating point numbers. Bit-serial processing involves identical processing of several consecutive bits, which means that pipelining on the bit level could probably be useful in the PEs.

The perfect shuffle/exchange structure is generally regarded as the most useful general purpose interconnection scheme for SIMD computers. With the size of the array fixed to 128 PEs, this was a reasonable choice for the interconnection network in LUCAS (a larger array would involve too much wiring of the back-planes). A data selector with eight inputs is used for defining the interconnection scheme. Two of these are used for the shuffle/exchange connection, two for the communication with neighbours where the PEs form a one-dimensional linear array, one for the X register and three are available for application dependent interconnections.

The input/output scheme is designed so as to let I/O and processing

be performed simultaneously. The sequential part of the transfer is entirely handled by the Host, which may read data from and write data into the I/O Registers of the PEs without affecting the execution of programs in the Associative Array. Transfer of data between the PE memories and the I/O Registers is performed in parallel in all PEs under microprogram control and takes only eight clock cycles per 128 bytes of data. In order to obtain a higher speed for the sequential data transfer, a special purpose I/O processor is now under development.

One of the most important parts of the Control Unit is the Address Processor. Its current implementation has proved to be very efficient for its main purpose, namely to provide bit addresses to the Associative Array. However, the microprograms could be made more powerful if the Address Processor included more functions, now handled by the Host. This would for example include facilities to handle complete descriptors to the logical fields of the Associative Array (see Section 7.3).

8.2.2 The Impact of VLSI Implementation

The regular structure of an associative array processor is very well suited for VLSI implementation. As part of a multi-project chip, the logic of one processing element (excluding memory) was implemented in CMOS/SOS by the project group in 1981. In this section we will discuss the possibilities of integrating many processing elements on one chip in terms of number of gate functions and number of pins per chip.

If we consider the consequences of using ordinary read/write memory chips for the memory modules, each processing element would need one input from above, one from below, one for shuffle and one for shuffle+exchange. We assume that the control signals for I/O data registers, multiplexer and ALU functions are contained in a single "instruction code", k bits wide. 8 bits would be appropriate to

implement the present possibilities. Finally, we assume b bits wide IO data registers. In LUCAS, b is 8.

Table 8.1 describes the number of pins needed on a chip comprising n processing elements.

<u>Specification</u>	<u>pins</u>
I/O data bus	b
I/O write	1
I/O data register address	$\log(n)$
Chip select	1
Select First chain	2
Data in/out	n
Shuffle input	n
Above/Below	2
Instruction code	k
Power, Ground, Clock	3

Sum: $b+k+9+2n+\log(n)$	

Table 8.1 The number of pins needed for an n processor chip.

The number of gate functions needed to implement the processors is totally dominated by the logic required to implement the arithmetic/logic unit. Assuming k_1 , out of the k instruction bits, are needed to specify the function and f flip flops in each PE,

$$G_1 = f * 2^{f+k_1+1}$$

gate functions are needed to implement the ALU of one PE as a ROM. (This number can be reduced considerably if a PLA is used instead of a ROM.) At present, we have $f=4$ and $k_1=5$, which means

$$G_1(LUCAS) = 4 * 2^{10} = 2^{12}$$

In an n processor chip, $n \cdot G_1$ gate functions are required.

We can now choose the parameters and a value of n that give values of gate count and number of pins within the limits of available technology.

For example

$$b=8$$

$$k=8$$

$$f=4$$

$$k_1=5$$

as in LUCAS, would make it possible to put 32 PEs in a 94 pin chip comprising 2^{17} (=128k) gate functions, which is quite feasible with current VLSI technology.

We assumed the memory modules were outside these chips. Suppose we want a memory word length of 64 kbits. We could then use memory chips of 64 K 8-bit words (the tag-masked write function would be more complex to allow selective writing of the bits). Four memory chips would be required to support one 32-PE chip of the above kind. A circuit board with 80 chips would then have 512 Processing Elements, each with 64 kbits of memory. Some additional chips for I/O address decoding and buffering would also be needed.

We next consider the inclusion of the memory modules in the PE chips. If we want to equip the PEs with index registers for addresses, this alternative must be chosen in order to limit the number of pins per chip. To provide a bit address to the memory, address pins are needed. We assume 2^m bits memory words, requiring m address bits. Furthermore, a write control signal is needed. Thus, the pin count exceeds what we had in Table 8.1 with $m+1$.

With $m=16$, the 16 processors/chip case would require 78 pins/chip. A board with 64 such chips would thus have 1024 processors in total.

The number of gate functions needed for the memory modules would dominate the gate function count in the chip. Assuming n processors

with 2^m bits memory each, $n \cdot 2^m$ gate functions are needed. For $n=16$ and $m=16$, this makes 2^{20} , which is one million. The gate count for the PE part ranges between 2^{12} and 2^{18} , depending on complexity.

We conclude that, with memory on the chip, it is probably the number of gate functions that puts the limit on how many processors can be implemented per chip.

8.3 LANGUAGE ASPECTS

8.3.1 Microprogramming

A language for microprogramming of LUCAS has been defined that allows microprograms to be written without a detailed knowledge of the internal workings of the machine. It provides the programmer with a simplified view of LUCAS, which comprises the PEs, the Common and the Mask Registers, and the I/O Buffer Register. The use of the language greatly simplifies the development of microprograms and also enhances the readability of programs, which facilitates documentation.

The language has a block structure where microprograms are contained in modules together with variables and subroutines they use in common. It features control constructs similar to those of high-level languages (if-then-else, while, repeat). An iterate-statement, which has a very natural application to bit-serial processing has been introduced in the language.

Experience from programming in microprogram assembly code has shown that the part of LUCAS which is the most difficult to program is the Control Unit, and great attention must be paid especially to the control of the Address Processor. The design of the language is such that the Address Processor disappears from the programmer's view and is entirely taken care of by the compiler.

A compiler for the language has been implemented. During the first

compilation pass, an intermediate code is generated in the form of purely sequential microcode. During the code improvement phase, a code compaction process is used which packs the sequential microoperations into complete microinstructions in which the potential parallelism of the horizontal microword organization is exploited. The code compaction algorithm is based on the First-Come First-Served algorithm [Das-1], which has been extended in several ways both in order to enhance its performance and also to provide a suitable model for the microprogrammed architecture of LUCAS.

8.3.2 Pascal/L

Pascal/L is defined in terms of extensions to standard Pascal. The extensions have been chosen so as to take full advantage of the processing capabilities of LUCAS. This means that typical SIMD operations, where one instruction operates on several data items in parallel, can be specified. The characteristic property of associative processing is the ability to designate a part of the data, which will be subject to parallel computation, in terms of properties of the data regardless of where it is stored. The indexing scheme of Pascal/L, where selectors may be used for referencing elements in the parallel variables, is the language counterpart of this associative principle.

The existence of a well defined high-level language, which is suitable for expressing parallel algorithms on LUCAS, has proved very useful. A compiler for Pascal/L has not yet been implemented, but the fact that Pascal/L, although general in its structure, is well adjusted for the capabilities of LUCAS, allows programs to be hand compiled for implementation on LUCAS.

The definition of a Pascal/L pseudo-machine suggests how the language could be implemented on LUCAS. By means of examples we discuss how constructs in Pascal/L can be executed on the pseudo-machine. An attractive possibility is to emulate the Pascal/L pseudo-machine on a standard computer, which would allow programs to

be tested before they were moved to LUCAS. By emulating the pseudo-machine, relevant performance estimations and extensive error-checking could be included.

Appendix 1

MICROPROGRAMMING INSTRUCTION SET

The microprogram word has 80 bits, which are logically divided into 24 fields. Some of the fields contain primary operations and must be specified in every line of the microcode. Others are parameter fields to some of the primary fields. Their value need only be specified when the corresponding primary field has certain values. Instead of presenting microprogramming in the special format used by the microprogram assembler (see [Kru-2]), we will describe the possible instructions in each field of the microprogram word.

In Table A1.1, primary fields are marked with '*' in the "type"-column. Parameter fields has the field number of its primary field in this column.

Field	Type	Bits	Function in LUCAS
1	*	1-4	Sequencer instruction
2	1	5-8	Select test input to the Sequencer
3	1	9-20	Sequencer data input
4	*	21-24	Address Processor instruction
5	4	25-28	Reg A address to the Address Processor
6	4	29-32	Reg B address to the Address Processor
7	4,13	33-44	Address Processor data input
8	*	45-46	Address Processor stack operation
9	-	47-51	Not used
10	*	52-53	Loopcounter instruction
11	*	54-55	Common Register and Mask Register write
12	*	56-57	I/O Buffer Register instruction
13	4	58-60	Select source to Address Processor data
14	*	61	NEXT. Acknowledge Host instruction
15	*	62	Load Status Register
16	*	63-64	"Signature". User defined function
17	*	65-69	PE ALU Instruction
18	17	70-72	PE ALU Multiplexer select
19	*	73-74	PE Memory write
20	*	75	Select First Tag
21	*	76	Network Enable
22	*	77	I/O Register shift
23	*	78	Select PE Memory data input
24	*	79-80	I/O Register read/write control

Table A1.1 Microinstruction fields

We will now proceed by giving a list of the possible instructions or parameter values of each field.

Field 1. Sequencer instructions

In the following description of the sequencer instructions, PL ("pipelineregister") refers to field 3 of the microprogram word.

CONT	CONTINUE. Execution continues with the next sequential microinstruction. This is the default instruction.
JZ	JUMP ZERO. The next microinstruction executed is at location zero in the microprogram memory.
JMAP	JUMP MAP. Jump to the address specified in the Instruction Register (the four most significant bits are set to zero). Used to start the execution of a microprogram.

- CJP** **CONDITIONAL JUMP.** Jump to the address specified in field 3 of the microprogram word, provided the test input is TRUE. Else continue with the next sequential instruction.
- CJV** **CONDITIONAL JUMP VECTOR.** If the test input is TRUE, a jump to the address provided in the I/O Buffer Register of the Control Unit will be taken (the four most significant bits are set to one). If the test input is FALSE then continue.
- JRP** **JUMP R/PL.** A jump is taken, either to the address in the internal Register/Counter (test input is FALSE) or to the address specified in field 3 (test input TRUE).
- CJS** **CONDITIONAL JUMP SUBROUTINE.** Perform a subroutine branch to the location specified in field 3, provided the test input has the value TRUE. Else continue.
- JSRP** **JUMP TO SUBROUTINE R/PL.** This instruction performs a sub-routine branch to either the address in the internal Register/Counter (test input FALSE) or to the address in field 3 of the microprogram word (test input TRUE).
- CRTN** **CONDITIONAL RETURN FROM SUBROUTINE.** If the test input is true then the next address is taken from top of the internal stack and the stack is POPed. Else continue.
- PUSH** **PUSH/CONDITIONAL LOAD COUNTER.** This instruction is used to set up loops in the microcode. The next sequential address is PUSHed on the internal stack (thus providing the loop-back address). If the test input is TRUE, the internal counter is also loaded with the value in field 3 (this specifies the number of times the loops should be repeated) If the test input is FALSE, the counter is not loaded.
- RFCT** **REPEAT LOOP, COUNTER <> ZERO.** This instruction is used for repeating a loop, which has been set up with the PUSH instruction. If the internal counter is non-zero, it is decremented and a branch back to the address found on top of the internal stack is performed. If the counter is zero, the stack is POPed and execution continues with the next sequential instruction.
- LOOP** **TEST END LOOP.** An alternative possibility to repeat a loop which has been set up with the PUSH instruction. In

this case, the test input is tested instead of the internal counter (which is not decremented). If the test input is FALSE, the branch back to the address on the stack top is taken. If the test input is TRUE, then continue.

- CJPP CONDITIONAL JUMP PL AND POP. This instruction is used to exit from inside loops which are set up with the PUSH instruction. When the test input is TRUE, a jump is taken to the address specified in field 3 and the stack is POPed. Else continue.
- LDCT LOAD COUNTER. The internal Counter/Register is loaded from field 3 of the microprogram word.
- RPCT REPEAT PIPELINE REGISTER, COUNTER $\neq$ ZERO. Similar to the RFCT instruction above, but the destination address is taken from field 3 rather than from the stack top. Is used to terminate loops, which have been set up with the LDCT instruction.
- TWB THREE WAY BRANCH. An instruction which tests both the test input and the internal Register/Counter:
- ```

 if test input is FALSE then
 begin
 if Register/Counter $\neq$ 0 then
 begin
 Decrement Register/Counter;
 Branch to top of stack;
 end else
 begin
 POP stack;
 Branch to address in field 3;
 end;
 end else
 begin
 POP stack;
 Continue;
 end;

```

Field 2. Select test input to the Sequencer

This field is used to select a status signal to be tested by the Sequencer's test input. Three bits are used to control selection in the Testmultiplexer and the fourth bit is used to complement the selected test condition.

|         |                                                                                                                                           |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------|
| ZAP     | ZERO ADDRESS PROCESSOR. True when the Address Register has the value ZERO.                                                                |
| NZAP    | Inverse to ZAP.                                                                                                                           |
| MASK    | True when the output from the Mask Register Memory has the value ONE.                                                                     |
| NMASK   | Inverse to MASK.                                                                                                                          |
| NZLC    | NON-ZERO LOOPCOUNTER. True when the Loopcounter in the control unit is non-zero.                                                          |
| ZLC     | ZERO LOOPCOUNTER. Inverse to NZLC.                                                                                                        |
| IL/BUSY | INSTRUCTION LOADED/BUSY. True when the Busy flip-flop is set. This indicates that the Instruction Register has been loaded from the Host. |
| NIL     | Inverse to IL/BUSY.                                                                                                                       |
| SOME    | True when at least one Tag register was set during the previous clock cycle.                                                              |
| NONE    | Inverse to SOME.                                                                                                                          |
| TRUE    | Always TRUE. (Default value).                                                                                                             |
| FALSE   | Always FALSE.                                                                                                                             |

### Field 3. Sequencer data input

This field has a value in the range 0 to 4095 (0-0FFF Hexa) and is used to define the value on the Sequencer data inputs.

### Field 4. Address Processor instruction

The Address Processor instruction set is listed in figure 2.6 of section 2.3.4. The Address Register serves as a pipeline register to the Address Processor and the instruction specifies the value that will be output during the next clock cycle.

### Field 5 and 6. Register specification to the Address Processor

Field 5 specifies the REGA address to the internal registers and field 6 the REGB addresses. Values are specified in the form REGA0, REGA1, ..., REGA9, REGAA, REGAB, ..., REGAF and REGB0, ..., REGBF.

### Field 7. Address Processor data input

Provided field 13 has the value "PIPELINE", this field is used to specify the data input to the address processor and to the Loopcounter. It has a value in the range 0 to 4095 (0-0FFF Hexa).

### Field 8. Address Processor stack operation

This field can specify either of the instructions SPUSH (STACK PUSH) or SPOP (STACK POP). SPUSH results in the value of the Address Register input being PUSHed on the stack (note that it is the current instruction to the Address Processor that decides the value to be PUSHed). SPOP pops the stack. If field 13 has the value "STACK", the POPed value is used as input to the Address Processor and the Loopcounter.

Field 10. Loopcounter instructions

The Loopcounter may be decremented with the instruction DECLC and loaded with the instruction LDLC, in which case it is loaded with the Address Processor input data (as selected by field 13).

Field 11. Common and Mask write

The instruction WRCOM and WRMASK writes the Common and Mask I/O Register values into the Common and Mask Register Memories.

Field 12. I/O Buffer Register instructions

LDIOBF loads the I/O Buffer with the value on the I/O Data Bus. This instruction is used in conjunction with instruction "RSEL" of field 24.

ENIOBF enables the I/O Buffer contents onto the I/O Data Bus. It is used together with instruction "IOWRALL" of field 24.

Field 13. Select source to Address Processor data input

This field specifies any of the following data sources to the Address Processor (and Loopcounter) data input:

|              |                                       |
|--------------|---------------------------------------|
| STACK        | Data from the Address Processor stack |
| PARAM1       | Parameter Register 1                  |
| PARAM2       | Parameter Register 2                  |
| PARAM3       | Parameter Register 3                  |
| PARAM4       | Parameter Register 4                  |
| IOBUFF       | The I/O Buffer Register               |
| PIPELINE/PPL | Field 7 of the microprogram word      |

Field 14. Next

When this field holds the instruction NEXT, the Busy flip-flop will be cleared, thus allowing the Host to send the next instruction.

Field 15. Load Status Register

The instruction LDSTAT enables the load function of the Status Register (see section 2.3.7).

Field 16. Signature

The meaning of these micro operations is user defined. The value is specified with the instructions S1,S2 and S3. They have for example been used when measuring the execution time of microprograms.

Field 17. PE ALU instructions

The following conventions are used in the instruction list below:

- \* Several of the instructions, which have been defined for the PE ALUs, exist in two versions; a tag-masked instruction (mnemonic ends with the letter T) which affects only the PEs where the Tag is set, and a non-tag-masked version (mnemonic ends with the letter A, for "all").
- \* Many functions have a couple of different mnemonics. For example CRZT (Compare R to Zero tag-masked) is the same function as LTRIT (Load Tag from R inverted tag-masked).
- \* In most cases, the X register is reloaded from the Multiplexer. This results in an unchanged value of X if the Multiplexer selects the X input.

- \* In arithmetic operations: R receives the result, C the carry and X receives the arithmetic overflow (used only when the sign bit has been processed). The previous value of C is used as incoming carry in the operation.
- \* The result of the COMPARE instructions goes to the Tag (T).
- \* ' indicates inverted value.
- \* The "Multiplexer" is abbreviated Mux.

| Type | Mnemonic | Function |
|------|----------|----------|
|------|----------|----------|

---

#### LOAD REGISTER

|       |                  |
|-------|------------------|
| LXMA  | Load X from Mux  |
| LTRIT | Load T from R'   |
| LTRT  | Load T from R    |
| LRMA  | Load R from Mux  |
| LRMT  | Load R from Mux  |
| LTMA  | Load T from Mux  |
| LTRA  | Load T from R    |
| LCRA  | Load C from R    |
| LRTA  | Load R from T    |
| LRCA  | Load R from C    |
| LTMT  | Load T from Mux  |
| LTMIT | Load T from Mux' |

#### EXCHANGE REGISTERS

|     |                  |
|-----|------------------|
| XRT | Exchange R and T |
|-----|------------------|

#### SET/RESET/COMPLEMENT REGISTER

|      |              |
|------|--------------|
| SCA  | Set C        |
| SETT | Set all tags |
| CCA  | Clear C      |
| CRA  | Clear R      |
| CORA | Complement R |
| CORT | Complement R |
| COT  | Complement T |

#### COMPARE

|      |                   |
|------|-------------------|
| CRZT | Compare R to Zero |
| CR0T | Compare R to One  |



|      |                       |
|------|-----------------------|
| CMOT | Compare Mux to One    |
| CMZT | Compare Mux to Zero   |
| CMCT | Compare Mux to Common |
| CRCT | Compare R to Common   |
| CRMT | Compare R to Mux      |

### LOGICAL FUNCTIONS

|         |                 |
|---------|-----------------|
| ANDTRIA | T AND R' to T   |
| ANDTRA  | T AND R to T    |
| ANDRMA  | R AND Mux to R  |
| ANDTMA  | T AND Mux to T  |
| ANDTMIA | T AND Mux' to T |
| XORRMA  | R XOR Mux to R  |
| XORRTA  | T XOR R to R    |
| ORRMA   | R OR Mux to R   |

### ARITHMETIC FUNCTIONS

|       |                          |
|-------|--------------------------|
| ADMA  | Add Mux to R             |
| ACMA  | Add Common to Mux        |
| ADMIA | Add Mux' to R            |
| SCMA  | Subtract Common from Mux |
| SUMA  | Subtract Mux from R      |
| ASMT  | ADD/SUB Mux to/from R    |
|       | where $T=1/0$            |

### Field 18. PE ALU Multiplexer select

These instructions decide which permutation should be performed over the Interconnection Network.

Note. In order to avoid noise on the Interconnection Bus, the Address Processor must output the bit address of the source during two clock cycles. Only during the second clock cycle (when data from the memories are stable), the "NETEN" instruction of field 21 is used, together with the applicable ALU instruction which loads the data from the Interconnection Network.

Possible instructions are:

|        |                                                                                        |
|--------|----------------------------------------------------------------------------------------|
| DIRECT | No permutation. Data comes from the PE's own memory. No extra addressing cycle needed. |
| X      | Data comes from the X register. No extra cycle                                         |

|        |                                                       |
|--------|-------------------------------------------------------|
|        | needed.                                               |
| SHUFF  | Data is shuffled.                                     |
| NSHUFF | Data is shuffled, then exchanged (neighbour shuffle). |
| ABOVE  | Data to PE no. $i$ comes from PE no. $i-1$ .          |
| BELOW  | Data to PE no. $i$ comes from PE no. $i+1$ .          |

#### Field 19. PE Memory write

Instructions are WR (write into PE memories, tag masked) and WRALL (write into all PE memories).

#### Field 20. Select First Tag

The instruction SELFI enables the Select First chain (see section 2.4.3)

#### Field 21. Network Enable

The instruction NETEN enables the PE memory output onto the Interconnection Bus. This function must only be activated when data is stable on the memory outputs - the same address must be generated by the Address processor during two clock cycles (see note under field 18)

#### Field 22. I/O Register Shift

The instruction IORS shifts the I/O Registers of all PEs, as well as those of the Common and the Mask Registers, one step to the right. Data on the PE memory output is shifted into the most significant bit of the register.

#### Field 23. Select PE Memory Data Input

By default the R register is used as input to the PE memory. The instruction SR selects data from the I/O Register's rightmost bit as data input.

#### Field 24. I/O Register Read/Write Control

The instruction RSEL enables the I/O Register contents of selected words onto the I/O Data Bus. To work properly, a previous "SELF" (field 20) must assure a unique selection of one PE. The instruction is used together with "LDIOBF" of field 12.

The instruction IOWRAL writes the value on the I/O Data Bus into all the I/O Registers. It is used in conjunction with "ENIOBF" of field 12.

## Appendix 2

### LUCAS Microprogramming Language

#### Syntax and Current PE Instruction Set

#### LUCAS Microprogramming Language Syntax in BNF

##### COMPILATION UNIT, MODULES AND MICROPROGRAMS

<compilation unit> ::=  
    <module>

<module> ::=  
    module <identifier>; <declaration part> <submodule part> endmod

<declaration part> ::=  
    <empty> | <constant declaration> |  
    <variable declaration> | <subroutine declaration> |  
    <constant declaration> <variable declaration> |  
    <constant declaration> <subroutine declaration> |  
    <variable declaration> <subroutine declaration> |  
    <variable declaration> <subroutine declaration>  
                    <subroutine declaration>

<submodule part> ::=  
    <empty> | <module>{;<module>} | <microprogram>{;<microprogram>}

<microprogram> ::=

microprogram <local declaration part> <statement part>

<microprogram heading> ::=  
microprogram <identifier>; |  
microprogram <identifier>(<formal microprogram parameter list>);  
 <formal microprogram parameter list> ::=  
 <microprogram parameter>{,<microprogram parameter>}<sup>3</sup>  
 <microprogram parameter> ::=  
 <empty> | <identifier>

## DECLARATIONS

<constant declaration> ::=  
const <identifier>=<sign><constant>{, <identifier>=<sign><constant>;}  
 <variable declaration> ::=  
var <identifier>{,<identifier>;}  
 <subroutine declaration> ::=  
 <subroutine heading><local declaration part><statement part>  
 <subroutine heading> ::=  
subroutine <identifier>; |  
subroutine <identifier>(<formal subroutine parameter list>;)  
 <formal subroutine parameter list> ::=  
 <identifier>{,<identifier>}  
 <local declaration part> ::=  
 <empty> | <constant declaration> | <variable declaration> |  
 <constant declaration> <variable declaration>

STATEMENT PART

<statement part> ::=  
     begin <statement list> end

<statement list> ::=  
     <statement>{,<statement>}

<statement> ::=  
     <empty> | <subroutine call> | <assignment> | <stack operation> |  
     <if statement> | begin <statement list> end | <exit statement> |  
     <loop statement>

<PE instruction> ::=  
     <PE0 identifier> | <PE1 identifier> (<address>) |  
     <PE2 identifier> (<address>,<permutation>)

<PE0 identifier> ::= <identifier>

<PE1 identifier> ::= <identifier>

<PE2 identifier> ::= <identifier>

<address> ::= <variable> | <constant>

<permutation> ::=  
     DIRECT | SHUFFLE | NSHUFFLE | ABOVE | BELOW

<subroutine call> ::=  
     call <identifier> | call <identifier>(<actual parameter list>)

<actual parameter list> ::=  
     <actual parameter>{,<actual parameter>}

<actual parameter> ::=  
     <variable identifier> | <sign><constant>

<assignment> ::=  
     <variable1>:=<sign><constant> | <variable2> |  
         <variable1><arithmetic operator><constant> |  
         <variable1><arithmetic operator><variable2>

<stack operation> ::=



SPUSH (<variable identifier>) | SPOP (<variable identifier>)

<if statement> ::=  
     if <condition> then <statement> |  
     if <condition> then <statement> else <statement>

<exit statement> ::=  
     exit (<loop identifier>)

<loop identifier> ::=  
     <identifier>

<loop statement> ::=  
     <loop label part> <while statement> |  
     <loop label part> <repeat statement> |  
     <loop label part> <iterate statement>

<loop label part> ::=  
     <empty> | <identifier> :

<while statement> ::=  
     while <condition> do <statement>

<repeat statement> ::=  
     repeat <statement list> until <condition>

<iterate statement> ::=  
     iterate <value> times <statement>

<condition> ::=  
     <variable><Boolean operator><variable> |  
     <variable><Boolean operator>0 |  
     TRUE | FALSE | SOME | NONE |  
     ZMASK(<address>) | NZMASK(<address>)

<constant> ::=   <number> | <constant identifier>

<value> ::=   <constant> | <variable identifier>

<constant identifier> ::=   <identifier>

<variable identifier> ::= <identifier>

<sign> ::= <empty> | -

<arithmetic operator> ::= + | -

<Boolean operator> ::= = | <>

## PE INSTRUCTION SET

The PE instructions embrace operations performed on the registers and on the memory in the Processing Elements. The instructions are of three kinds:

Without parameters. These instructions use the PE registers as operands and leave the result in the registers.

With one parameter. These instructions either uses the Common Register as an operand or store the R register in the PE memory. The parameter specifies the PE memory address.

With two parameters. These are instructions where one of the operands comes from the interconnection network. The first parameter gives the PE memory address of the source bit. The second parameter specifies the permutation of data over the network.

The PE instruction set may be changed by reprogramming the ALU PROMs. The instruction list below describes the current instruction set. In the list the following conventions are used:

- \* Several of the instructions exist in two versions: A tag-masked instruction (the instruction name ends with the letter "T"), which affects only the selected PEs. A non-tag-masked instruction (the name ends with the letter "A", for "all"), which affects all PEs.
- \* Arithmetic operations: R receives the result, C the carry

and X the arithmetic overflow (used only when the last bit has been processed). The previous value of the C register is used as incoming carry.

- \* The result of the compare instructions affect the Tag in the following way: A Tag which has the value "zero" is not affected. A Tag which is "one" gets the value "zero" if the compare fails.

## PE INSTRUCTIONS WITHOUT PARAMETERS

### Load/Exchange Register

|       |                        |
|-------|------------------------|
| LTRA  | Load T from R          |
| LTRT  | Load T from R          |
| LTRIT | Load T from R'         |
| LTXA  | Load T from X, T to X  |
| LTXT  | Load T from X, T to X  |
| LTXIT | Load T from X', T to X |
| LRTA  | Load R from T          |
| LRCA  | Load R from C          |
| LRXA  | Load R from X, R to X  |
| LRXT  | Load R from X, R to X  |
| LCRA  | Load C from R          |
| XRT   | Exchange R and T       |

### Set/Reset/Complement Register

|      |                                                                                                 |
|------|-------------------------------------------------------------------------------------------------|
| STA  | Set T                                                                                           |
| SCA  | Set C                                                                                           |
| CCA  | Clear C                                                                                         |
| CRA  | Clear R                                                                                         |
| COTA | Complement T                                                                                    |
| CORA | Complement R                                                                                    |
| CORT | Complement R                                                                                    |
| SELF | SELECT FIRST. Clear T in all PEs with number > i, where PE no. i is the first PE where T is One |

### Compare

|      |                   |        |
|------|-------------------|--------|
| CRZT | Compare R to Zero |        |
| CROT | Compare R to One  |        |
| CRXT | Compare R to X    | T to X |
| CXOT | Compare X to One  | T to X |
| CXZT | Compare X to Zero | T to X |

Logical

|         |               |        |
|---------|---------------|--------|
| ANDTRA  | T AND R to T  |        |
| ANDTRIA | T AND R' to T |        |
| ANDRXA  | R AND X to R  | R to X |
| ANDTXA  | T AND X to T  | T to X |
| ANDTXIA | T AND X' to T | T to X |
| ORRXA   | R OR X to R   | R to X |
| XORRTA  | R XOR T to R  |        |
| XORRXA  | R XOR X to R  | R to X |

Arithmetic

|       |                                   |
|-------|-----------------------------------|
| ADXA  | Add X to R                        |
| ADXIA | Add X' to R                       |
| ASXT  | Add/Subtr X To/From R where T=1/0 |
| SUXA  | Subtr X from R                    |

PE INSTRUCTIONS WITH ONE PARAMETER

The parameter specifies a bit address to the PEs.

|       |                             |
|-------|-----------------------------|
| WRRRA | Write R into PE memory      |
| WRRT  | Write R into PE memory      |
| CRCT  | Compare R to Common         |
| CXCT  | Compare X to Common, T to X |
| ACXA  | Add X to Common             |
| SCXA  | Subtr Common from X         |

PE INSTRUCTIONS WITH TWO PARAMETERS

The first parameter specifies a bit address to the PEs. The second parameter specifies a permutation on the bit-slice, which is performed before the data enters the PE ALU. In the instruction list below, "M" is used to denote incoming data from the Interconnection Network.

Possible permutations are:

|          |                                                                                             |
|----------|---------------------------------------------------------------------------------------------|
| DIRECT   | No permutation                                                                              |
| SHUFFLE  | The bit-slice is shuffled (see section 1.6.2)                                               |
| NSHUFFLE | The bit-slice is shuffled, then exchanged                                                   |
| ABOVE    | The bit-slice is rotated one step down. Data to PE no.i comes from PE no. $(i-1) \bmod 128$ |
| BELOW    | The bit-slice is rotated one step up                                                        |

### Load/Exchange Register

|       |                |        |
|-------|----------------|--------|
| LRMA  | Load R from M  | R to X |
| LRMT  | Load R from M  | R to X |
| LTMA  | Load T from M  | T to X |
| LTMT  | Load T from M  | T to X |
| LTMIT | Load T from M' | T to X |
| LXMA  | Load X from M  |        |

### Compare

|      |                     |        |
|------|---------------------|--------|
| CMOT | Compare M to One    | T to X |
| CMZT | Compare M to Zero   | T to X |
| CMCT | Compare M to Common | T to X |
| CRMT | Compare R to M      | T to X |

### Logical

|         |               |        |
|---------|---------------|--------|
| ANDRMA  | R AND M to R  | R to X |
| ANDTMA  | T AND M to T  | T to X |
| ANDTMIA | T AND M' to T | T to X |
| ORRMA   | R OR M to R   | R to X |
| XORRMA  | R XOR M to R  | R to X |

### Arithmetic

|       |                                     |  |
|-------|-------------------------------------|--|
| ADMA  | Add M to R                          |  |
| ADMIA | Add M' to R                         |  |
| ACMA  | Add M to Common                     |  |
| ASMT  | Add/Subtr M To/From R where $T=1/0$ |  |
| SUMA  | Subtr M from R                      |  |
| SCMA  | Subtr Common from M                 |  |

## Appendix 3

### Pascal/L - SYNTAX IN BNF

#### DATA DECLARATIONS

<selector type> ::=  
    selector[<range>] |  
    selector[<range>] := <Boolean aggregate>

<range> ::=  
    <constant>..constant

<Boolean aggregate> ::=  
    <choice> => <Boolean value>

<choice> ::=  
    <constant> | <constant>..constant |  
    <constant>..constant step <constant>

<Boolean value> ::=  
    true | false

<parallel array type> ::=  
    parallel array[<range>] of <parallel type> |  
    parallel array[<range>,<constant>..constant]  
        of <parallel type>

<parallel type> ::=  
    <parallel type identifier> | <parallel standard type> |  
    record <parallel field list> end

<parallel type identifier> ::=  
    <identifier>

<parallel standard type> ::=  
    integer(<constant>) | unsigned integer(<constant>) | Boolean |  
    fixed(<constant>.<constant>) | char | string(<constant>)



```

<parallel field list> ::=
 <parallel record section> {,<parallel record section>}

<parallel record section> ::=
 <field identifier> {,<field identifier>} : <parallel standard type>

<field identifier> ::=
 <identifier>

```

## MICROPROGRAM DECLARATION

```

<microprogram declaration> ::=
 microprogram <identifier> <microprogram parameter list>; external;

<microprogram parameter list> ::=
 <empty> |
 <microprogram parameter>{,<microprogram parameter>}

<microprogram parameter> ::=
 <empty> | <identifier>

```

## INDEXING

```

<indexed parallel variable> ::=
 <parallel variable identifier> |
 <parallel variable identifier>[<first index>] |
 <parallel variable identifier>[<first index>,<expression>]

<parallel variable identifier> ::=
 <identifier>

<first index> ::=
 * | <constant> | <constant>..<constant> | <selector expression>

```

STATEMENTS

<where statement> ::=

where <selector expression> do <statement> |

where <selector expression> do <statement> elsewhere <statement>

<parallel case statement> ::=

case where <parallel expression> of

        <case list element>{; <case list element>;} <others part> end

<case list element> ::=

    <case label>{,<case label>} : <statement>

<others part> ::=

    <empty> | others : <statement>

<while and where statement> ::=

while and where <selector expression> do <statement>

## References

- [Aho-1] Aho, A.V., Ullman, J.D. "Principles of Compiler Design", Addison Wesley Publishing Comp., 1979.
- [And-1] Anderson, G.A. "Multiple Match Resolutions: A New Design Method", IEEE Trans. on Computers, Dec. 1974.
- [Bab-1] Baba, T., Hagiwara, H. "The MPG System: A Machine-Independent Efficient Microprogram Generator", IEEE Trans. on Computers, Vol C-30, No. 6, June 1981.
- [Bar-1] Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A. "The ILLIAC IV Computer", IEEE Trans. on Computers, Vol. C-17, pp. 746-757, Aug. 1968.
- [Bat-1] Batcher, K.E. "The Flip Network in STARAN", Proc. of the 1976 Int. Conf. on Parallel Processing, 1976.
- [Bat-2] Batcher, K.E. "The Multidimensional Access Memory in STARAN", IEEE Trans. on Computers, Feb. 1977.
- [Bat-3] Batcher, K.E. "The STARAN Computer", Infotech State of the Art Report Supercomputers, Infotech Intl. Ltd., Maidenhead, Berks., UK, 1979.
- [Bat-4] Batcher, K.E. "Bit-Serial Parallel Processing Systems", IEEE Trans. on Computers, Vol. C-31, No. 5, May 1982.
- [Bat-5] Batcher, K.E. "Architecture of a Massively Parallel Processor", The 7th Annual Symposium on Computer Architecture, La Baule, France, 1980.
- [Bra-1] Bratsbergseugen, K., Risnes, O., Amble, T. "ASTRAL - A Structured and Unified Approach to Data Base Design and Manipulation", RUNIT Comp. Center at the University of Trondheim, Norway. Report No. STF14.A80003, 1979.
- [Cim-1] Cimsa Corp. "Propal2 - Processeur Parallele Associatif", Cimsa - Compagnie d'Informatique Militaire Spatiale et Aeronatique, 1979. Cimsa, 10-12 avenue de l'Europa,

Velizy, France.

- [Das-1] Dasgupta, S., Tartar, J. "The Identification of Maximal Parallelism in Straight-Line Microprograms", IEEE Trans. on Computers, Vol. C-25, No. 10, Oct. 1976.
- [Das-2] Dasgupta, S. "Some Aspects of High-Level Microprogramming", Computing Surveys, Vol. 12, No. 3, Sept. 1980.
- [Dav-1] Davidson, S., Landskov, D., Shriver, B.D., Mallett, P.W. "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Trans. on Computers, Vol C-30, No. 7, July 1981.
- [DeW-1] DeWitt, D.J. "A Machine-Independent Approach to the Production of Horizontal Microcode", Ph. D. thesis, Univ. of Michigan, Ann Arbor, June 1976.
- [DeW-2] DeWitt, D.J. "Extensibility - A New Approach for Designing Machine-Independent Microprogramming Languages", Proc. 9th Ann. Workshop on Microprogramming (ACM), Sept. 1976.
- [Fei-1] Feierbach, G.F., Stevenson, D.K. "The Phoenix Array Processor", Proc. 17th Ann. Tech. Symposium, June 1978.
- [Fel-1] Feldman, J.A., Rovner, P.D. "An Algol-Based Associative Language", Comm. of the ACM, Vol. 12, No. 8, Aug. 1969.
- [Fer-1] Fernström, C. "Programming Techniques on the LUCAS Associative Array Computer", Proc. of the 1982 International Conf. on Parallel Processing, Aug. 1982.
- [Fer-2] Fernström, C., Kruzela, I., Ohlsson, L., Svensson, B. "An Associative Parallel Processor Used in Real Time Signal Processing", To appear in the Proc. of Second European Signal Processing Conference, Erlangen, W.-Germany, Sept. 1983.
- [Fer-3] Fernström, C. "The LUCAS Associative Array Processor and its Programming Environment", PhD thesis, Dept. of Computer Engineering, University of Lund, Sweden, 1983.
- [Fin-1] Findler, N.V. "On a Computer Language which Simulates Associative Memory and Parallel Processing", Cybernetica (Namur) Vol. X, No. 4, 1967.

- [Fis-1] Fisher, J.A., Landskov, D., Shriver, B.D. "Microcode Compaction: Looking Backward and Looking Forward", Proc. 1981 National Computer Conf., AFIPS 1981.
- [Fla-1] Flanders, P.M. "DAP-FORTRAN Language", CM39, RADCL, ICL 1976.
- [Flo-1] Floyd, R.W. "Algorithm 97: Shortest Path", Comm. of the ACM, Vol. 5, p. 345, 1962.
- [Fly-1] Flynn, M.J. "Very High-Speed Computing Systems", Proc. of the IEEE, Vol. 54, No. 12, Dec. 1966.
- [Fos-1] Foster, C.C. "Content Addressable Parallel Processors", Van Nostrand Reinhold Co., 1976.
- [Gol-1] Golumb, S.W. "Permutations by Cutting and Shuffling", SIAM Rev., Vol. 3, pp. 293-297, Oct. 1961.
- [Gri-1] Gries, D. "Compiler Construction for Digital Computers", John Wiley & Sons, Inc., 1971.
- [Hig-1] Higbie, L.C. "Supercomputer Architecture", Computer, Dec. 1973.
- [Hoc-1] Hockney, R.W., Jesshope, C.R. "Parallel Computers", Adam Hilger Ltd, Bristol, 1981.
- [Jen-1] Jensen, K., Wirth, N. "Pascal User Manual and Report", Springer-Verlag, 1974.
- [Kru-1] Kruzela, I.K., Svensson, B.A. "The LUCAS Architecture and its Application to Relational Data Base Management", Sixth Worksh. on Comp. Architecture for Non Numerical Processing, Hyeres, France, Jun. 1980.
- [Kru-2] Kruzela, I.K. "LUCAS Microprogramming Manual", Internal Report, Dept. of Computer Engineering, University of Lund, Sweden, Dec. 1980.
- [Kru-3] Kruzela, I.K. "An Associative Array Processor Supporting a Relational Algebra", PhD thesis, Dept. of Computer Engineering, University of Lund, Sweden, 1983.
- [Kru-4] Kruzela, I.K. Personal Communication.
- [Kuc-1] Kuck, D.J. "ILLIAC IV Software Application Programming", IEEE Trans. on Computers, Vol. C-17, No. 8, Aug. 1968.

- [Kuc-2] Kuck, D.J. "The Structure of Computers and Computations - Volume One", John Wiley & Sons Inc., 1978.
- [Lan-1] Landskov, D., Davidson, S., Shriver, B., Mallett, P.W. "Local Microcode Compaction Techniques", Computing Surveys, Vol. 12, No. 3, Sept. 1980.
- [Lan-1] Lang, T. "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network", IEEE Trans. on Computers, Vol. C-25, No. 5, May 1976.
- [Lan-2] Lange, R.G. "High Level Language for Associative and Parallel Computation with STARAN", Proc. of the 1976 International Conf. on Parallel Processing, 1976.
- [Law-1] Lawrie, D.H. "Access and Alignment of Data in an Array Processor", IEEE Trans. on Computers, Vol. C-24, No. 12, Dec. 1975.
- [Law-2] Lawrie, D.H. "Glypnir Programming Manual", ILLIAC IV Doc. No. 232, ILLIAC IV Proj., University of Illinois at Urbana-Champaign, Urbana, Ill.
- [Law-3] Lawrie, D.H., Layman, T., Baer, D., Randal, J.M. "Glypnir - a Programming Language for ILLIAC IV", Comm. of the ACM, Vol. 18, No. 3, March 1975.
- [Lay-1] Layman, T., Baer, D. "Glypnir Reference Manual", ILLIAC IV Doc. No. 263, ILLIAC IV Proj., University of Illinois at Urbana-Champaign, Urbana, Ill.
- [Lev-1] Levialdi, S., Isoldi, M., Uccella, G. "Programming in PixaI", IEEE Workshop on Picture Data Description and Management, Asilomar, California, Aug. 1980.
- [Lov-1] Love, H.H. "Programming the Associative Linear Array Processor", Proc. of the 1975 Sagamore Comp. Conference on Parallel Processing.
- [Mez-1] Mezzalama, M., Prinetto, P., Filippi, G. "Microcode Compaction via Microblock Definition", Proc. 15th Ann. Workshop on Microprogramming (ACM), 1982.
- [Mic-1] Mick, J., Brick, J. "Bit-Slice Microprocessor Design", McGraw-Hill Book Company, 1980.



- [Mil-1] Millstein, R.E., Muntz, C.A. "The ILLIAC IV FORTRAN Compiler", Proc. of a Conf. on Programming Languages and Compilers for Parallel and Vector Machines, March 1975.
- [Mil-2] Millstein, R.E. "Control Structures in ILLIAC IV FORTRAN", Comm. of the ACM, Vol. 16, No. 10, Oct. 1973.
- [Mue-1] Mueller, P.T. Jr, Siegel, L.J., Siegel, H.J. "A Parallel Language for Image and Speech Processing", Proc. of the COMPSAC 80, Oct. 1980.
- [Orc-1] Orcutt, S.E. "Efficient Data Routing Schemes for ILLIAC IV - type Computers", Digital Systems Lab., Stanford University, Tech. Rep. 70, Apr 1974.
- [Ozk-1] Ozkarahan, E.A., Schuster, S.A., Smith, K.C. "RAP - An Associative Processor for Data Base Management", National Computer Conference 1975.
- [Par-1] Parker, D.S. "Notes on Shuffle/Exchange-Type Switching Networks", IEEE Trans. on Computers, Vol. C-29, No. 3, Mar. 1980.
- [Pat-1] Patterson, D.A. "Strum. Structured Microprogram Development System for Correct Firmware", IEEE Tran. on Computers, Vol. C-25, No. 10, Oct. 1976.
- [Pea-1] Pease, M.C. "An Adaptation of the Fast Fourier Transform for Parallel Processing", Journ. of the ACM, Vol. 15, pp. 252-264, Apr. 1968.
- [Per-1] Perrott, R.H. "A Language for Array and Vector Processors", ACM Trans. on Prog. Languages and Systems, Vol. 1, No. 2, Oct. 1979.
- [Pre-1] Presberg, D.L., Johnson, N.W. "The Paralyzer: IVTRAN's Parallelism Analyzer and Synthesizer", Proc. of a Conf. on Programming Languages and Compilers for Parallel and Vector Machines, March 1975.
- [Ram-1] Ramamoorthy, C.V., Tsuchiya, M. "A High-Level Language for Horizontal Microprogramming", IEEE Trans. on Computers, Vol. C-23, No. 8, Aug. 1974.
- [Red-1] Reddaway, S.F. "The DAP Approach", Infotech State of the Art Report Supercomputers, Infotech Intl. Ltd.,

Maidenhead, Berks., UK, 1979.

- [Ree-1] Reeves, A.P., Bruner, J.D., Poret, M.S. "The Programming Language Parallel Pascal", 1980 Internat. Conf on Parallel Processing.
- [Ree-2] Reeves, A.P., Bruner, J.D. "High Level Language Specification and Efficient Function Implementation for the MPP", Internal Purdue Electrical Engineering Report TR-EE 80-32, Jul. 1980.
- [Ree-3] Reeves, A.P., Bruner, J.D., Brewer, T.M. "High Level Languages for the MPP", Internal Purdue Electrical Engineering Report TR-EE 81-45, Nov. 1981.
- [Res-1] Resnick, H.K., Larson, A.G. "DMAP - A COBOL Extension for Associative Processors", Proc. of a Conf. on Programming Languages and Compilers for Parallel and Vector Machines, March 1975.
- [Sch-1] Schomberg, H. "A Peripheral Array Computer and its Applications", Parallel Computers - Parallel Mathematics, M. Feilmeier (ed.), Internat. Assoc. for Mathematics and Computers in Simulation, 1977.
- [Slo-1] Slotnick, D.L. "The Conception and Development of Parallel Processors - a Personal Memoir", Ann. of the History of Comp. Vol. 4, No.1, Jan. 1982.
- [Ste-1] Stevens, K.G. Jr. "CFD - A FORTRAN-Like Language for the ILLIAC IV", Proc. of a Conf. on Programming Languages and Compilers for Parallel and Vector Machines, March 1975.
- [Sto-1] Stone, H.S. "Parallel Processing with the Perfect Shuffle", IEEE Trans. on Computers, Vol. C-20, Feb. 1971.
- [Sve-1] Svensson, B., Ohlsson, L. "Matrix Multiplication on LUCAS", Internal Report, Dept. of Computer Engineering, University of Lund, Sweden, Jan. 1983.
- [Sve-2] Svensson, B. "Image Operations Performed on LUCAS - an Array of Bit-Serial Processors", to appear in the Proc. of the 3rd Scandinavian Conf. on Image Processing, Copenhagen, Denmark, 1983.

- [Sve-3] Svensson, B. "LUCAS Processor Array - Design and Applications", PhD thesis, Dept. of Computer Engineering, University of Lund, Sweden, 1983.
- [Thu-1] Thurber, K.J. "Large Scale Computer Architecture", Hayden Book Comp., Rochelle Park, New Jersey, 1976.
- [Tok-1] Tokoro, M., Tamura, E., Takizuka, T. "Optimization of Microprograms", IEEE Trans. on Computers, Vol. C-30, No. 7, July 1981.
- [Tur-1] Turner, D.A. "Error Diagnosis and Recovery in One-Pass Compilers", Information Processing Letters, 6, 4, pp. 113-115.
- [Uhr-1] Uhr, L. "A Language for Parallel Processing of Arrays, Embedded in Pascal", Comp. Sciences Technical Report £365, Sept. 1979.
- [Wir-1] Wirth, N. "The Design of a PASCAL Compiler", Software-Practice and Experience, 1, No. 4, 1971.
- [Yau-1] Yau, S.S., Schowe, A.C., Tsuchiya, M. "On Storage Optimization for Horizontal Microprograms", Proc. 7th Ann. Workshop on Microprogramming (ACM), 1974.











Christer Fernström

## THE LUCAS ASSOCIATIVE ARRAY PROCESSOR AND ITS PROGRAMMING ENVIRONMENT

The LUCAS project is an attempt to design and to evaluate a highly parallel system. As part of the project an SIMD organized Associative Array Processor comprising 128 bit-serial processors has been implemented. This thesis reports on the design of LUCAS and of languages suitable for implementation on this and similar designs.

A high-level like language for microprogramming has been defined and implemented on LUCAS. The compiler, which produces efficient horizontal microcode, is described. For application programming, a high level language has been defined in terms of extensions to Pascal (Pascal/L). The extensions needed to express parallel processing in an SIMD organized computer include new data types and new constructs in the control structure.

Two related theses report further design issues programming and application studies:

Bertil Svensson: Associative Array Processor and Algebra, PhD thesis 1983.

Bertil Svensson: LUCAS Processor Array – Design and Applications, PhD thesis 1983.

